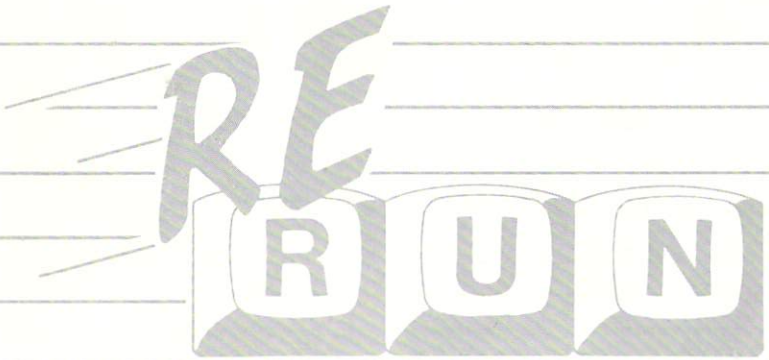
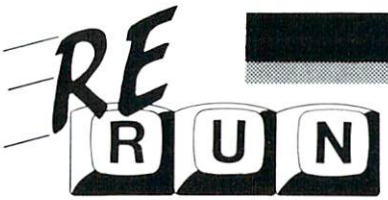


REPRINTED ARTICLES FROM JANUARY TO JUNE 1984 RUN MAGAZINE



Please send me RERUN VOLUME II! I understand that it will be available with the December 1984 issue of RUN.

____ **Cassette version(s) at $11.47* each.**
____ **Disk version(s) at $21.47 each.**

* Prices include $1.50 postage and handling.
Foreign Air Mail please add an additional 45¢ per item.
U.S. funds drawn on U.S. banks ONLY.

☐ **Check/MO** ☐ **MC** ☐ **VISA** ☐ **AE**

Card #_____ Exp. Date_____

Signature_____

Name_____

Address_____

City_____ State_____ Zip_____

RERUN • 80 Pine Street • Peterborough, NH • 03458

DIRECTORY OF PROGRAMS

INTRODUCTION		3
OPERATING INSTRUCTIONS		6

COMMODORE 64

"ZELAZ64	JAN P42"	8
"SYM-CODE	JAN P92"	15
"KINGDOM	FEB P76"	17
"DATABASE	FEB P48"	19
"FNCTKEY	FEB P70"	21
"SPRITEN	FEB P124"	25
"BOMBER	MAR P106"	27
"TLMUSIC	MAY P132"	30
"REPEAT	MAY P82"	32
"DSK-O-64	JUN P54"	35

"UP C64><DOWN VIC"*

VIC 20

"DSKOVIC	JAN P102"	38
"KNGDM	FEB P76"	17
"FUNCTKEY	FEB P70"	21
"DBASE/3K	FEB P48"	19
"VICASSO	FEB P132"	46
"DOODL/3K	JUN P98	49
"SERP/3K	MAR P58"	54
"SERPENT MODULE"		54
"BAHA1000	MAR P68"	58
"BAHA MODULE"		
"CAVES	MAY P90"	61
"CAVES MODULE"		
"FUNKEY	APR P58"	62

The file names listed above contain the month of issue and page number.

*This program is not included on the cassette version of RERUN. Cassette users please note that Side A contains the Commodore programs and Side B contains the VIC 20 programs.

program is loaded, type RUN and press the return key, and you're on your way.

If you're a disk user, just type

```
LOAD "entire-program-name" , 8
```

then press the return key and the program should load.

When we say "entire-program-name," we mean *either* the entire name, including the month and page number, just the way it appears on the box, or the "wild-card" method, the first few letters followed by an asterisk.

For example, if you want to load the program DISK-O-VIC, you must type it *exactly* as we show it on the box—DSKOVIC JAN P102—or, if you're lazy (like me), you can type the first four or five letters of the name and add an asterisk (*) as a wild card. So to load DISK-O-VIC, you would only have to type

```
LOAD "DSKO*" , 8
```

(check your manuals for more on how to load programs).

We could go on and on about RE-RUN—how wonderful the programs are, how inexpensive, how easy, how to load and run them, how much work we put into this and so on, but all you have to do is order a copy, try a few of the programs and see for yourself. After all, if you didn't think that *RUN* magazine was worthwhile, then you wouldn't be reading this now, would you?

GW

How to load programs from RERUN:

DISK—
To load any of the programs type:

LOAD " program-name " , 8

then press the RETURN key.
The disk drive should 'whirr' while the screen prints SEARCHING FOR program-name. The screen should then print LOADING and then finally READY with the flashing cursor beneath. Type RUN and press the RETURN key. The program will then begin.

CASSETTE—
Insert the cassette tape into the Datasette recorder with the proper side facing up (Commodore 64 side up if you own a Commodore 64 and VIC-20 side up if you own a VIC-20)
Make sure that the tape is rewound all the way to the beginning.
Type

LOAD " program-name "

then press the RETURN key. The screen will display
PRESS PLAY ON TAPE
you should then push the play button on your datasette recorder. WARNING: do not press the RECORD button and the PLAY button at the same time or you may destroy the programs on the tape.

The datasette motor should then start by itself. When the program has been found the screen will display
FOUND program name
on some Commodore computers you may then have to press the C= (Commodore symbol) key to then load the program. On other Commodore machines the program will load automatically. Check your owner's manual for specific loading procedures.
When the program has finished loading you will see the READY prompt and the flashing cursor beneath. Type RUN and press the RETURN key to start the program.

NOTE: 1
You should use the entire program name as listed to avoid loading programs that have similar titles.
NOTE 2:
Make sure that if you are loading VIC-20 programs you have the correct memory expansion cartridge (or no cartridge if that is required) plugged in before loading the program. The memory configurations are listed as part of the title. EG: "DBASE/3K FEB P48" requires a 3K memory expansions cartridge.
NOTE 3:
Some VIC-20 programs are divided into two sections, the main section (the one you should load first) and the MODULE section that is either automatically loaded when the first section is run or is loaded manually after the first section is run.

IMPORTANT
Commodore 64 programs (the first 10 programs on the disk) will NOT normally run on a Commodore VIC-20 and by the same token VIC-20 programs will NOT usually run on a Commodore 64. Even though you may be able to load a particular program into the wrong computer it is unlikely that it will run properly.
ALWAYS refer to the article in the magazine (month and page numbers are given in the title of each program) for operating instructions, memory requirements, etc.

CANYONS OF ZELAZ

You're good—no doubt about it. You've fought off hundreds of invaders, made the jump to hyperspace and shot your way through a meteor swarm that was blizzard-thick on your sensors. Now you've earned a rest, unless . . . maybe you're good enough to fly the mail run in the Canyons of Zelaz, a game that illustrates the use of sprite graphics on the Commodore 64.

By Gary D. McClellan

Enjoyable games can be created in Basic using the C-64's sprite graphics capability—without the need for writing routines in assembly language. The simple process for generating such a game can be broken down into three general steps:

1. Define the scenario and what action will take place.
2. Create the graphics images necessary to complete the scenario.
3. Write the program.

Scenario

Writing a game program is similar to writing a short story; the background and setting are important. Since I've always been fond of lunar-lander-type games, I decided a lander game using sprite graphics would be fun to write. I wanted a different setting than Earth's moon, however, so I decided on the following scenario.

RUN It Right

Commodore 64
Joystick

Zelaz is an airless planetoid discovered in 2183. Mineral deposits of commercial quantity were located in the northern hemisphere. The first three landing parties perished when their ships were destroyed at landing. Fluxes in the planetoid's magnetic field were discovered, and a landing team was sent in farther south.

After a successful landing, the team moved north with tracked vehicles. Rich deposits of Ellisonite were discovered. Mining operations began and the excavations followed the Ellisonite plugs down through the crystalline mantle of Zelaz.

The aberrations in the magnetic field stabilized below the surface of Zelaz, so the miners nicknamed the destructive forces at the top of the canyons the "magwinds." A warning beacon was erected at the top of each canyon wall to warn of the danger. Other problems occurred.

The crystalline layer surrounding the deposits of Ellisonite reacted violently to earth-manufactured alloys. To protect their equipment and themselves from serious damage, the miners left a thin coating of Ellisonite covering the canyon sides as a buffer zone. A heavily shielded shuttle was built to haul loads of Ellisonite from the bottom of the subcanyons created by the mining operation to the landing pad at the upper level of the main canyon.

Action

The mission of the shuttle pilot is to fly into each subcanyon and to land successfully at the bottom. At each of three landing pads in the canyons, the shuttle drops off mail and supplies for the mining team working there and takes on a cargo of Ellisonite and 300 units of fuel. The shuttle then must be flown to the next landing pad or return to the upper landing pad, where greater supplies of

fuel are available.

If the shuttle makes contact with a landing pad at a velocity of −10 or less or a horizontal velocity of 4.5 or greater or −4.5 or less, the shuttle will explode. If the shuttle brushes against the canyon walls briefly, nothing will happen. More than brief contact will explode the ship.

When the shuttle is returning to the upper pad, if the shuttle altitude reaches the same altitude of the warning beacons, the shuttle will be ripped apart by the magwinds or hurled into the upper canyon walls.

After all three lower landing pads have been reached and the shuttle has safely returned to the upper pad, the mission is complete. A status report will be generated on the mission computer and the shuttle pilot scored.

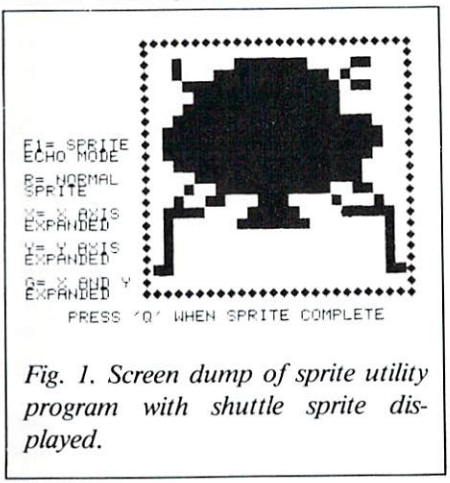


Fig. 1. Screen dump of sprite utility program with shuttle sprite displayed.

Graphics

To complete the scenario, the graphics images required were made up of a shuttle craft, landing pads and the canyon.

The shuttle craft is a perfect candidate for sprite graphics, since a complete graphics image needs to be moved around the screen rather quickly. The image of the shuttle (see Fig. 1) was created using the sprite utility program for the C-64 from the June 1983 issue of *Microcomputing*. The shuttle craft image then was edited to add a rocket exhaust (see Fig. 2). The 63-byte data description for each image was displayed and then entered into data statements in the game program (see Figs. 3 and 4).

Since the shuttle would be interacting with the landing pads, a sprite image of a pad was created. This was done by filling the first 24 bytes of the 63-byte image with decimal value 255. The resulting graphics image is a solid block, eight pixels deep by 24 pixels wide.

The canyon was created by poking values for the C-64's low-resolution graphics characters into screen memory.

Program Description

After defining the scenario and creating the graphics images to go along with it, you'll write the program. In the debugging process, you'll discard a few ideas and find others that will modify the scenario and the graphics. The creative process continues until you're satisfied with the results.

In the completed game program, let's look at the beginning lines, then at the various subroutines and finally at the main program loop to see how we can

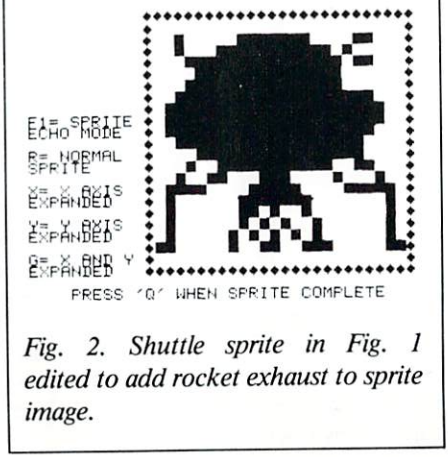


Fig. 2. Shuttle sprite in Fig. 1 edited to add rocket exhaust to sprite image.

animate the sprite images to obtain the results we need.

Lines 5–30: The program clears the screen in line 5 and then prints the starting message.

Line 60: Line 60 sets up the game before the program enters the main loop at line 100. The variable 0F is an offset value between the standard screen memory starting at location 1024 and screen color memory starting at location 55296. Anytime a value is stored into screen memory, the color can be set by storing a color value into SCREEN LOCATION + 0F.

Subroutines at line 500 and line 900 are called to draw the background for the game; they enable the sound generator and sprite graphics.

Lines 500–597: The routine at line 500 generates the game background and

```
BYTE 1      0           0           0
BYTE 4      32          254         24
BYTE 7      35          255         160
BYTE 10     31          255         216
BYTE 13     15          255         192
BYTE 16     31          255         192
BYTE 19     63          255         248
BYTE 22     127         255         252
BYTE 25     127         255         252
BYTE 28     127         255         248
BYTE 31     63          255         240
BYTE 34     15          255         240
BYTE 37     19          255         152
BYTE 40     38          56          68
BYTE 43     120         124         62
BYTE 46     64          254         2
BYTE 49     64          0           2
BYTE 52     64          0           2
BYTE 55     64          0           2
BYTE 58     192         0           3
BYTE 61     0           0           0
ENTER ANY CHARACTER FOR MENU.
?
```

Fig. 3. Sixty-three-byte data description of shuttle sprite displayed in Fig. 1.

mission computer display and loads the sprite images into memory. Data values are read from the data statements beginning at line 1000 and stored into three 64-byte sections of memory.

```
BYTE 1      0           0           0
BYTE 4      32          254         24
BYTE 7      35          255         160
BYTE 10     31          255         216
BYTE 13     15          255         192
BYTE 16     31          255         192
BYTE 19     63          255         248
BYTE 22     127         255         252
BYTE 25     127         255         252
BYTE 28     127         255         248
BYTE 31     63          255         240
BYTE 34     15          255         240
BYTE 37     19          255         152
BYTE 40     38          56          68
BYTE 43     120         124         62
BYTE 46     64          254         2
BYTE 49     64          210         2
BYTE 52     64          169         2
BYTE 55     65          85          2
BYTE 58     193         51          131
BYTE 61     0           0           0
ENTER ANY CHARACTER FOR MENU.
?
```

Fig. 4. Sixty-three-byte data description of shuttle sprite with exhaust displayed in Fig. 2.

A sprite is defined by 63 bytes of data, but the 64th byte is used by the C-64 as a control byte for each image. The C-64 is set up to handle up to eight sprite images at a time, so a value has to be assigned to each sprite image defined. This value is the number of the memory section in which the sprite image is stored. The sprite images defined by the data statements are stored in memory sections 13, 14 and 15 ($13 \times 64 = 832$, the starting address at which the data is stored).

After the values in the data statements have been read and stored into memory, we have three sprite image definitions we can use. The landing pad image is in section 13, the shuttle image in section 14, and the shuttle with exhaust is in section 15.

Line 504 dimensions a string array and creates four elements of the array that will be used in the mission status report.

Line 506 clears the screen, and then sets the screen background color to

dark gray and the screen border color to light gray. The For...Next loop fills screen color memory with the value for the color white.

Lines 508–510 draw the border for the mission computer on the right-hand side of the screen and color the border an off-world purple.

Lines 512–577 draw the game background on the screen. Since color memory already has been filled with white, the character graphics poked to the screen will appear as white objects against a dark gray background.

Lines 584 to 597 draw the instrument readouts for the mission computer on the screen.

Lines 900–950: This subroutine enables the sprite images previously placed in memory and initializes the sound generator.

In line 900, the starting address of the C-64 video controller is set to variable V. Variable X is loaded with the beginning X-axis coordinate for the shuttle. The variable Y is loaded with the beginning Y-axis coordinate for the shuttle. Variable D is set to 3 and used in the main program loop to calculate altitude. Memory location V + 31 contains the sprite-to-background collision register and is initialized to 0. Variable A2 is the starting altitude of the shuttle when the game begins, and variable FU will be used to count the number of times the shuttle refuels during the game.

Images Enabled

Sprite images are enabled for use by setting from one to eight bit values to 1 in the sprite-enable register at location V + 21. Setting a bit to 1 will turn on a sprite. The first bit in the register is sprite 0, the second bit enables sprite 1, and so on (bit 0 is also the first bit in a byte). By poking the value 31 into the enable register, we set bits 0 through 4 to 1, and enable five sprites.

The C-64 needs a way to determine which sprite image will be used by an enabled sprite. Eight locations at the end of screen memory have been reserved for this function. Memory locations 2040–2047 correspond to sprites 0–7. By poking the value 15 into location 2040, we define the sprite image at section 15 as sprite 0.

At this point, five sprites have been enabled and sprite 0 has been defined. Sprites are positioned on the screen by loading an X and Y coordinate into the sprite position registers at locations V through V + 15. The X coordinate for sprite 0 is at location V. The Y coordinate for sprite 0 is at location V + 1.

Now that the shuttle is enabled, defined and positioned on the screen, a value of 1 is poked into location V + 27. This location is the sprite/background priority register. By setting bit 0 (which represents sprite 0) to 1, the shuttle sprite will disappear "behind" any low-resolution graphics characters it encounters on the screen.

Sprites 1–4 are defined as the sprite image in section 13 in line 916. This is the landing pad image. Since we need four landing pads, we can define four different sprites using the same basic image. The X and Y coordinates for the four landing pad sprites are set in line 918.

Memory locations V + 39 through V + 46 contain color registers for sprites 0–7. Line 920 pokes the value for light gray into the register for the shuttle, and pokes the value for green into the registers for the landing pad sprites. Line 92 sets bits 1, 2, 3 and 4 to 1 in sprite expand X register at location V + 29. This doubles the horizontal size of the landing pads.

Locations V + 37 and V + 38 are two extra color registers used with multicolor sprites. These two locations are loaded with the value for yellow and red

and will be used by the shuttle crash routine.

Line 930 sets the registers of the sound generator to 0 with a For...Next loop and then defines variables for waveform, attack/decay, sustain/release, note frequency low and note frequency high. The high- and low- frequency values for a note are then poked into the registers at line 940.

Lines 300–395: The mission report routine consists of three separate subroutines that are called when a shuttle crash occurs, a mission abort is requested or the mission is successfully completed.

The routine is entered at line 300 in a crash sequence. Line 300 stores note values in memory and then sets the volume control register at location 54296 to high. Attack/decay, sustain/release and waveform are loaded with values, and the noise begins.

The variable CK is the crash flag and is set to 1. In line 304, the sprite image for sprite 0 (this previously was the shuttle) is defined as memory section 11, a blank section of memory. A value of 1 is poked into the multicolor sprite select register at location V + 28. Sprite 0 is now a multicolor sprite.

Fireworks

The explosion begins in line 305. The subroutine at line 380, which fills 25 random locations in memory section 11 with random values, is called. Sprite 0 is then expanded on the X axis, and subroutine 380 is called again to add more random multicolor points to sprite 0.

Sprite 0 is then expanded on its Y axis, and once again subroutine 380 is called to provide a changing color effect. The explosion is complete and the subroutine at line 390 is called to clear memory section 11 for later use. Line 310 turns off the noise and the explosion is over.

The mission report status is printed in line 315, and line 317 tidies up the registers by clearing the expanded X and Y coordinates of sprite 0. This disables sprite 0 in the enable register and turns off the multicolor mode. The program then goes to line 370 and samples joystick port 2 for input. If the fire button is pressed, the program reinitializes and returns to the calling routine. If the joystick handle is pulled down and the fire button pressed, the program turns off the sprites; clears the screen, variables and pointers; and ends the program.

Line 350 is the mission-abort routine. The mission status is printed, and then the program goes to line 370 to wait for joystick input to restart or end the program.

Lines 360–363 make up the mission-complete routine. A score is calculated for the mission based on the number of refueling stops, and a mission report is then displayed. The program executes the routine at line 370 and restarts or ends the program.

Lines 400–496: The shuttle/pad collision routine is called whenever the shuttle sprite is in physical contact with a landing pad sprite. When this occurs, if the vertical velocity is less than −10 or the horizontal velocity is greater than 4.5 or less than −4.5, line 400 calls the crash subroutine at line 300. Line 403 checks if the shuttle has touched the upper landing pad, and calls the crash routine at line 300 if the shuttle is not lined up on the pad.

At line 405, the shuttle has not met the crash requirements, so we have a successful landing. A mission-status message is displayed, the shuttle sprite is defined as the image without exhaust, and noise from the rocket is turned off.

If the shuttle has landed on the upper landing pad, line 410 checks to see if all three lower pads have been visited. A subroutine at line 495 is called. If all

lower landing pads are red, then variable Q is set to 1. The subroutine returns to line 410, and if Q is equal to 1, the program goes to the mission-complete routine at line 360. If the mission is not complete, 200 extra units of fuel are added to the shuttle in line 415.

The Y coordinate is updated after a landing by line 435. Line 440 prints a status report update if the program has returned from the crash routine. If the shuttle has landed on one of the three lower pads, lines 450–455 refuel the shuttle and change the color of the landing pad the shuttle is on from green to red, and update the refueling counter. The upper pad is then colored green in case it has changed to red from a refueling.

The mission-status instrument readouts are updated in lines 460–471; the program then waits for the fire button to be pushed for take-off. If the joystick handle is pulled down and the fire button pressed, the mission is aborted. When take-off is initiated, line 485 sets vertical and horizontal velocity, clears the sprite-to-sprite collision register and resets the shuttle altitude. The mission status is updated in line 490 and the shuttle is once again in flight.

Lines 100–200: The main program loop controls the game while the shuttle is in flight. Line 100 sets collision flags CS and CP to 0, resets the sprite-to-background collision pointer and reads the value of joystick port 2. If the fire button is not pushed, line 105 sets the vertical acceleration variable and turns off the sound of the rocket engine.

When the fire button is pressed, lines 107–113 handle the rocket routines. Line 107 sets the sound volume to high and turns on the sound of the rocket. The fuel total is lessened, deacceleration is set and sprite 0 is defined as the graphics image in memory section 14 (shuttle with exhaust).

The program then checks to see if the joystick is pushed left, right or up. When it's left or right, horizontal velocity is incremented or decremented in lines 110 and 112. When the joystick is pushed forward, line 113 increments the fuel variable and sets the acceleration variable to hold velocity at a constant value.

Line 150 changes the color of the warning beacons at the top of each side of the canyon to red, and then calculates velocity and altitude. Line 155 checks the sprite/background collision register at location V + 31 to determine if the shuttle is in contact with the canyon walls. Collision counter CR is reset to 0 if the shuttle is clear of the walls. The X and Y coordinates of the shuttle are calculated in line 176. If the shuttle is too high in altitude and being affected by the magwinds, horizontal velocity is increased.

Line 177 is the in-flight crash check. If the shuttle has been hurled into the far right- or left-hand wall, or is high enough into the magwinds to be destroyed, the program calls the crash routine at line 300.

Line 178 clears the sprite/background collision register and pokes the shuttle coordinates into the X and Y position registers of sprite 0. The sprite/background collision register is rechecked, and, if a collision has occurred, the collision counter is incremented by 1. If the collision counter is greater than 1, the collision flag CS is set to 1.

The mission computer readouts are updated in lines 180–186. Line 190 checks for a sprite/sprite collision, and calls the shuttle/pad collision routine at line 400 if required. Line 195 calls the crash routine at line 300 if the crash flag is set; otherwise, the beacon colors are turned to yellow and sprite 0 is defined

as memory section 15.

The program then jumps to line 100 and continues the loop.

Game-Playing Hints

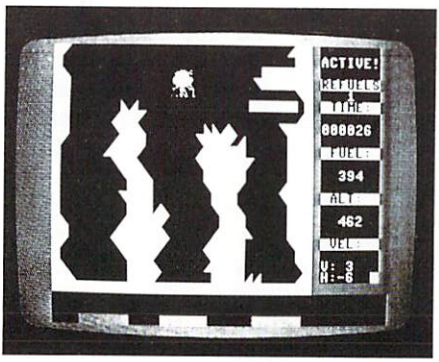
After the game program is entered into your machine, you're ready to fly a mission.

Until you get the feel of the shuttle, be careful when taking off. Positive vertical velocity builds up rapidly and you'll find yourself in the magwinds. Push the joystick to the left as you take off to build up horizontal velocity to help you clear the landing pad. The shuttle's on-board computer will hold the horizontal velocity constant until increased or decreased by the joystick.

The lower pads are green until you land; then they turn red. If you reland on a red pad, you won't receive any fuel, since it already has been depleted.

If at first you are burning too much fuel, take off and then immediately reland on the upper pad. Each time you reland, your fuel reserves will build. Your final score will suffer, but you'll be able to complete the mission.

If you find yourself on a lower pad and feel you don't have enough fuel to continue, pull back on the joystick and the mission will be aborted.



Conclusions

Enjoyable games can be written in Basic without using assembly-language routines. The sprite graphics capabilities built into the Commodore 64 let the programmer control hi-res graphics images easily. Canyons of Zelaz can be further modified and enlarged by adding your own routines to it.

The purpose of Canyons of Zelaz was to provide an example of how to use sprite graphics in game scenarios. I hope this prompts you to enjoy the game, modify and change it and then write your own game and send it to *RUN*. R

For more on the Commodore 64's sprite graphics capabilities, see *Microcomputing*, June 1983, p. 60—"Sprites, Graphic Eyes and the C-64."

Address author correspondence to Gary D. McClellan, PO Box 346, Rimrock, AZ 86335.

SYMBOL CODE

This game is a mind challenger. Fast thinking is a plus, but fast reaction, manual dexterity, and everything else that makes you a good arcade-game player, are irrelevant.

By Evangelos Petroutsos

Symbol Code is an adaptation for the Commodore 64 computer of the once very popular table-game, Mastermind. The object of Mastermind is to break a code consisting of a sequence of four colors selected from a palette of six different colors. The code is set by another player. Each time you make a guess, you are provided with some information concerning the success of your attempt. Good judgment and the use of all available information will help you break the hidden code.

Fig. 1. The symbols used to form the codes in the game program, Symbol Code.

RUN It Right

Commodore 64

Address author correspondence to Evangelos Petroutsos, 851 Camino Pescadero #70, Goleta, CA 93117

Many Permutations

Symbol Code uses six different symbols (Fig. 1) instead of colors, so that the display is more interesting and the game can be played on either a color or a black and white TV set. Since the code consists of four out of six symbols, there are at least 360 possible four-symbol codes. If we allow any symbol to appear more than once in the code, there can be as many as 6^4, or 1296, possible codes. The challenge is to break the hidden code in as few trials as possible. Your task is not easy, as you will discover!

Let's take a look at the game. After you run the program, an empty grid will appear on the screen. Its columns are labeled 1, 2, 3, 4, FG and PS. The six symbols are displayed at the right-hand side of the screen, with an arrow pointing to the first one. Your guesses will appear in the first four columns.

The columns labeled PS and FG will be filled in by the computer as follows: FG is the number of symbols you have guessed correctly. PS shows the number of correctly guessed symbols which are also in the correct position.

For example, three dots in the FG column and one dot in the PS column indicate that in your last guess you have guessed three symbols correctly, but only one of them is in the correct position. Your task is to find out which are the correctly guessed symbols and which one is in the correct position.

The grid is large enough to display six consecutive efforts. If you have not broken the code after the six trials, your next guess will replace the least recent one. After you become familiar with the game, however, you will realize that you don't really need more than six guesses to break the code.

Making the Moves

To enter your move, first, choose a symbol and the position in which you want it placed. Then, using the function keys, move the arrow up (F1) or down (F7) so that it points to the desired symbol. Lastly, hit the key (1, 2, 3 or 4) corresponding to the square in which you wish to place the selected symbol, and it will appear there.

In case you change your mind, you can overwrite any symbol in any position, changing the combination as many times as you wish before hitting the space-bar to enter your move. But once all four squares are filled and you hit the space-bar, there is no turning back. When you hit the space-bar, the program will read your move, compare it to the hidden code and display the results of the comparison in the FG and PS columns next to your guess.

Although the hidden code ordinarily consists of four *different* symbols, you may choose to repeat a symbol in any given guess to try to find out whether or not it belongs to the hidden code. Beginners frequently use this technique, which sometimes—combined with a little luck—provides useful information about the hidden code at the first stages of the game.

Suppose, for example, that the hidden code does *not* include the symbol "x" or "+," and that your first guess is "+ + x x." The results of the comparison tell you immediately the four symbols making up the code. Then you have to determine their order. After breaking the hidden code, you can start a new game by pressing any function key.

One last feature of the game is the Help command. As you try to break the hidden code, four small black squares are continuously displayed at the lower right-hand corner of the screen. These are the symbols in the hidden code. Each time you press H, one of the symbols in the hidden code will be revealed. When you break the code, all four symbols will be displayed there.

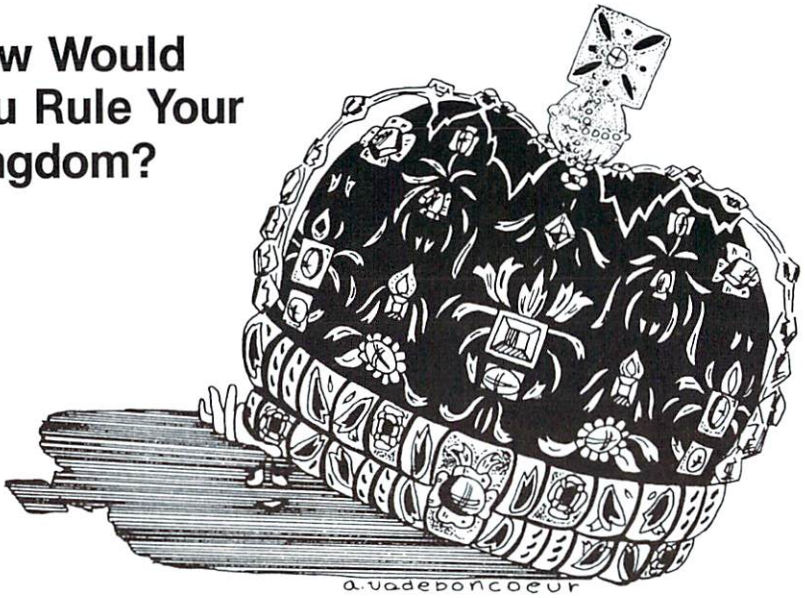
A word about the score. The maximum score you can get in one play is 500. If you break the code with your first or second guess, you get all 500 points. If not, each unsuccessful guess after the first two will cost you ten points. You also lose 50 points every time you ask for Help, and asking for Help four times in the same play reduces your score to zero. Your current score, as well as the average score of all your previous games, will be displayed at the top of the screen.

Listing 1 shows you the game program, along with some useful remarks. Lines 135 and 137 compare the symbols in the hidden code to make sure no symbol appears twice. If you find the game too easy to play, you can remove these two lines. Doing so will almost triple the number of possible combinations for the hidden code.

A word of caution: This program plays around with the display list, and you should be very careful when typing it into your computer. Any typos in the Poke statements might cause the computer to crash. Be sure you Save the program before you try to run it, so that even if something goes wrong, you'll not have to type it in again. R

Iron Hand Or VIC-20?

How Would You Rule Your Kingdom?



As a royal ruler in this game, you've got to think quickly to expand your kingdom while forestalling grain drain, especially in time of war.

By Joseph J. Shaughnessy

RUN IT RIGHT

VIC-20 or Commodore 64

Address author correspondence to Joseph J. Shaughnessy, 4703 Country Club Drive, Pittsburgh, PA 15236.

This game is both fun and educational. You must continually juggle numbers and computations in your head, but it's not a painful process. (The program will work with any memory configuration of the VIC-20, including unexpanded, and also with the C-64. See Listings 1 and 2.)

You are the ruler of a small city-state in ancient times. Your major goal in life is to increase the size of your kingdom, and you measure your progress towards this goal by the number of acres that you own. To be successful, you'll find that caring and concern for the people under your rule may not always be productive in accomplishing your goal. However, total disregard of your people also carries penalties.

Royal Decisions

As the game begins, you own land, have grain in storage and also have a population to govern. During each

round of play (measured as one year for each round), you must buy or sell land, set aside grain for feeding the population during the year and determine how many acres to plant.

There are many factors to ponder in planning for the coming year. Is the state at war or peace? Is there sufficient food to feed the population, or should some of the people be allowed to starve? Are there enough people to do the work of planting and harvesting—and for military service if there is war? Is the price of land high or low? Have you saved enough grain for seed? Are there any fringe benefits with this job?

To aid you with your job, the State of the Realm report is constantly displayed and updated as you are requested to give orders for the upcoming year. Also, at the end of each year, the Grand Secretary of State will give you a report of the results of your decisions, including such things as harvest yield, census changes and the state of the treasury.

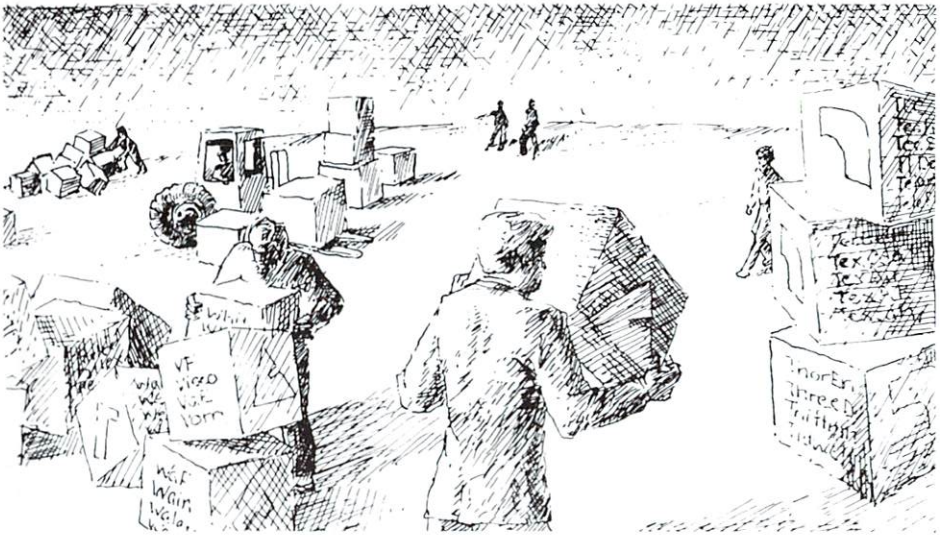
You could discover, through trial and error, the requirements for distributing grain to your various priorities, but that is maddening. Instead, I will tell you the following: each person requires 20 bushels of grain to eat; each person can only plant 10 acres of land; seed requirements are ½ bushel per acre.

In time of war, one-third of the population is required for the army and is not available for agriculture (they still eat, though). You will find yourself at war about 30 percent of the time.

Variables Used

A list of the variables used in this program is as follows: p = population; ar = new arrivals to the city; sp = number of people who starved; rd = grain destroyed by rats; yh = harvest yield in bushels per acre; gh = total bushels harvested; gs = current bushels in storage; ca = acres owned by the kingdom; yr = current year of your reign; pw$ = war or peace; a$, i$, i, j, zz, x = temporary variables; wk = number killed in war; wr = flag for warehouse raid; wf = population efficiency factor for war or peace; k = current price of land; b$ = increase or decrease in kingdom size. R

DATABASE DELUXE



Need an inexpensive data-base program for your VIC-20 or C-64? This one won't cost you a cent. Just type in the listing to store, categorize and sort your data with ease.

By John Stilwell

Deluxe File Case is a file handler for the Commodore 64 or for the VIC-20 with a memory expansion of 3K or more. The program is designed to use the 1540 or 1541 single disk drive or the Commodore Datassette. For printouts, it will work with any of the VIC printers.

The file format is a group of pages with ten entries per page. In the VIC-20 version, line 30 looks at the amount of memory available for data storage and then gives you the optimum number of pages. This means that if you change the size of the program, it will notice and will change the number of pages it gives you.

In the C-64 version, you are always given 100 pages with ten entries on each page. On line 30, N is set to 1000, the number of entries that the file can hold. If you want more or fewer pages, all you have to do is change this number.

When you run the program, you will first be asked for a file name. If you push the return key without providing a name, the file name will default to "Noname." The program then sets itself up.

A moment later, the list of one-letter commands will appear on the screen. The commands that you have are: Page, Insert, Enter, Catalog, Alpha-

RUN It Right

VIC-20 with 3K or more of expanded memory, or Commodore 64
1540 or 1541 disk drive or
Commodore Datassette

Address author correspondence to John Stilwell, 5018 Marathon Drive, Madison, WI 53705.

betize, Kill, New, Load, Save, Hard Copy and Help. If you should ever forget what they mean, push the ? key for the list of definitions.

Using the Commands

To call up a page, push P and the page number that you want. (A flashing cursor will remind you to push the return key after typing in something that was asked for.) When the page appears, you will see ten entry numbers with a dash after each one. To make an entry, push E and type in one of the numbers to indicate where you want the entry to go. The entry must not contain any commas, colons or semicolons. After you have pushed the return key, the entry will appear on the page.

If you want the entry to appear in the catalog, it has to be reversed (lettering inside a colored bar). To do this, the first character of the entry must be a left arrow. This is the key in the upper left-hand corner of the keyboard.

I reverse such things as the titles of categories. For example, you might want to organize a book list by authors. To do this, reverse each author's name and enter his books after the name. (The book titles are *not* reversed.) Now, whenever you call the catalog, each author's name will be shown with the page number on which it appears.

To insert something between two already existing entries, push I and type the number of the line that you want the insertion to go on. If you want to kill (erase) an entry, push K and type in the entry number. To cancel a command like Kill, just type in another command letter instead of the entry number.

If you want to skim over pages, push the space bar instead of a command letter. To call up the catalog, push C. Due to the limited amount of space on the screen, only ten categories with their page numbers can be displayed at a time. Push the return key to get the next ten categories.

To save or load a file, push S or L respectively. You'll be asked to confirm your intentions. You wouldn't want to load in a file when you're not yet done with the one that's in the computer. Next, you'll be asked whether you're using a disk or tape drive. Push D for disk or T for tape. If you push D, you'll be asked if you want a listing of the data files that are on the disk.

When resaving a file to the disk, the old one will be replaced by the new one. This relieves you of trying to remember which version of a file is the most recent one.

Push A to alphabetize. You can only alphabetize what is inside of a category. If your categories are authors' names, then you can alphabetize the books by each individual author, but you cannot alphabetize the authors themselves. After pushing A, you will be asked for the number of the first entry to be sorted. R

FUNCTION KEYS

If you've been wanting a way to define the VIC-20 function keys to your own needs, here's a program that lets you do it with ease.

By John Tanzini

When you first purchased your VIC-20, you undoubtedly wondered about the function keys. You may have been disappointed the first time you pressed one and found that nothing happened.

I can remember searching through the reference manual to determine how to assign functions. I found that the function keys are very easy to use in Basic programs, since they can be input like any other character, but I had hoped for more. I had hoped to be able to assign functions that would aid me in programming—functions that would execute as if they were part of the operating system.

There is a way. If you print a predefined string to the screen every time a function key is pressed, you can execute any function with a single keystroke.

For example, suppose the word LIST is printed when F1 is pressed. Now press the return key, and your program will be listed. If the return key had been defined as part of the string, then simply pressing F1 would list the program. Similarly, F2 could be made to run a program.

I find, while debugging certain programs, that I am constantly typing PRINT PEEK (N), where N is the number of some memory location. Since cursor controls can be included in strings, I can define a string which prints PRINT PEEK ( ) and then moves the cursor back to the position just after the left parenthesis. Then all I have to do is type the number of the location that I wish to interrogate, and press the return key.

You will doubtless have your own idea of what functions should be assigned to the function keys. It is a simple matter to customize my program to your own needs and define the keys and way you like.

Although part of this program is written in machine language, you need no knowledge of machine language to use the program or to redefine the function keys. So read on and put those function keys to work for you.

RUN It Right

VIC-20
Assembler

Using the Program

A copy of the program is shown in Listing 1. It is not as long as it appears, since you do not have to type in any of the REM statements. Be sure to save a copy of the program before you try to run it.

The most likely place to make a mistake while entering the program is in the Data statements, which contain the machine language program. For that reason, I have included a checksum at the end of each Data statement. The last number of each Data statement is the sum of all the previous numbers in that line.

When the Basic program loads the machine language program, it checks the checksum in each statement. If it does not add up properly, the program assumes that one or more of the numbers in that line was incorrectly typed, and an error message is then printed. The error message tells you exactly which line is incorrect, which should aid you considerably in getting the program running.

When you run the program, you should see a list appear on the screen, showing exactly how the function keys are defined. A few seconds will pass while the machine language program is loading, and then READY will appear on the screen. The Basic program should have automatically cleared itself out of memory by executing a New.

At this time you should be able to use the function keys. Pressing F1, for example, will print the word LIST. Functions F9 through F12 are obtained by pressing the Commodore key and one of the function keys.

Understanding one point about the operation of the Basic program will help you get the program running. The first thing the program does is move the top-of-memory pointer way down to protect a block of memory where the machine language program will reside.

If you have made a typing error in a Data statement, the program will detect it when calculating the checksum and will branch to line 800. At line 800 the program will restore the top-of-memory pointer, which returns all of the memory back to the operating system. If it did not, the program wouldn't have sufficient memory to execute correctly the next time you tried to run it.

If, however, you enter a Basic statement incorrectly (causing a syntax error), the operating system will stop the program immediately, without restoring anything. If you execute a GOTO 800 right after the program stops, you will save yourself the trouble of turning your VIC off, then on again, and reloading the program. Of course, this problem will not occur once the program is entered as shown in the listing.

You will find that pressing the run/stop and restore keys deactivates the program. This is because the operating system restores the interrupt vector to its original value. The program can be restarted by simply executing a SYS 0.

After you are sure the program is running properly, you may remove lines 611 through 618, along with the last data item in each Data statement. That is the part of the program associated with the checksum. The machine language program will load in about half the time with the checksum removed. Do not forget also to remove the last comma in each Data statement.

Since the machine language program remains in memory after the Basic program clears itself out, you will lose a small amount of memory. Your free memory will decrease by 144 bytes, plus one byte for every character defined in your strings.

Redefining the Function Keys

The function keys F1 through F12 are defined in lines 210 through 320. An array of strings named F$ holds a string associated with each function key. F$(1) is the string defined for F1; F$(2) is the string assigned to F2, and so on. To redefine a function key, simply change the appropriate line of the program corresponding to the function key that you wish to change.

For example, line 210 defines the string for F1:

```
210 F$(1) = "LIST"
```

If, instead, you would like the word LOAD to be printed when F1 is pressed, change line 210 to:

```
210 F$(1) = "LOAD"
```

Be sure to include the quotes, since F$ is a string variable.

Any valid string can be assigned to the function keys, including strings containing cursor controls. There are, however, two characters that are slight-

ly more complicated to assign within a string. They are the Return and the Quote. To include a Return in a string, add CHR$(13) to the string (13 is the ASCII code for Return). For example, if you want F3 to automatically start running a program as soon as you press the key, change line 230 to:

```
230 F$(3) = "RUN" + CHR$(13)
```

The return will be executed immediately after printing RUN, just as if you had pressed the return key on the keyboard. A quote can be included in a string in a similar manner using CHR$(34).

The maximum total length of all the strings you assign to the function keys is 231 characters. If you assign more than 231 characters, the program will print out an error message indicating that your strings are too long. At that time you may simply edit the appropriate lines and run the program again.

Keep in mind that the program clears itself out of memory after it runs. So if you would like to have a permanent copy of the program with your newly defined functions, remember to save the program before you run it.

The following is a brief description of how the machine language section of the program works. For a commented assembly-language listing of the program, send an SASE to *RUN* magazine.)

The general technique used to activate the function keys is fairly simple. Sixty times every second, a hardware interrupt is generated that signals the operating system to perform certain housekeeping functions such as scanning the keyboard and updating the real-time clock. By intercepting this interrupt, the machine language program executes sixty times a second.

Every time the program executes, it checks to see if one of the function keys is pressed. If one is pressed, the keyboard buffer is loaded with as many characters of the appropriate string as it can hold. As soon as the keyboard buffer is emptied by the operating system, my program will load more of the string into the buffer, until the string is completely printed.

The Basic Program

The functions of the Basic program are: to load the machine language program at the top of memory; to load the strings just below the machine code; to set up pointers for the machine language program; and to protect program and strings from the rest of the operating system.

Lines 100 to 130. Reserve enough memory to load the machine language program and strings by changing the top-of-memory pointer to point 512 bytes above the Basic program.

Lines 200 to 350. The array F$ is created, and the strings associated with each function key are printed to remind the user how they are defined.

Lines 400 to 450. The total length of all the strings is calculated. It is verified that their length does not exceed 231 characters. If the strings are valid, then SM (start of machine language program) and SS (start of strings) are calculated.

Lines 500 to 530. The strings and a table of pointers to the strings are loaded, beginning at location SS.

Lines 600 to 630. The machine language program is read from the Data statements and is loaded, beginning at location SM.

Lines 700 to 780. A pointer to the machine language setup routine is stored in memory. The top of memory is changed to point to the beginning of

the strings, so that only as much memory as is needed is taken away from the operating system. The program jumps to the machine language setup routine, then executes a New.

Lines 800 to 820. Execution reaches this point only if an error occurs, such as defining strings that are too long. The top of memory is restored to its original value in order to return all the memory to the operating system before stopping.

Lines 1000 to 1080. This is the subroutine that takes a string F$(I) and loads it into memory. A pointer to the string is also loaded into a table.

Conclusion

The power of this program lies in the fact that you can customize it to your own needs. If you have a printer, for example, one of the keys can be defined to give you a printout with a single keystroke. Some of the commands I have defined are useful only if you have a disk drive. If you define the set of functions that you use the most, you will find this program very handy. R

Address author correspondence to John Tanzini, Wynbrook West Apt. O-8, Dutch Neck Road, E. Windsor, NJ 08520.

Take the tedium out of programming sprite graphics. This C-64 program simplifies the process.

By Edward Rager

The capacity to create and manipulate sprites is a powerful feature of the Commodore 64. However, there's a lot of work involved in doing it. Probably the most tedious aspect of sprite graphics is translating the binary data from the sprite you draw into decimal numbers that can be Poked into memory.

The program described here allows you to draw an enlarged version of your sprite on the screen. The computer will scan the diagram, calculate the numbers to be Poked into memory and display your sprite.

How to Draw a Sprite

The C-64 user's guide gives a detailed description of how to create a sprite. Essentially, you fill in the spaces of a grid. A 1 goes in a space you want to have filled in, and a 0 goes in a space to be left blank. There are 21 rows and 24 columns. The 24 columns are divided into three 8-bit binary words.

Run It Right

Commodore 64

Address author correspondence to Edward Rager, 9360 Tasmania Ave., Baton Rouge, LA 70810.

So 21 rows, composed of three 8-bit words each, make 63 words that describe your sprite. When converted into decimal values and Poked into memory, the sprite can be displayed on the screen.

Once you have entered the program, typing RUN will draw the sprite borders on the screen. (There won't be any grid lines.) The program will stop here to let you draw a sprite within the borders. Use the cursor arrows to move the cursor to a position you want filled in and put a 1 there. It is not necessary to put a 0 in spaces you want left blank, for the computer looks only for 1s.

When the drawing is complete, type GOTO200: with the cursor at the left margin of the screen and about halfway from the top. (In typing in these program commands, be sure to include each colon. If any are omitted, syntax errors will result.) Your drawing will be scanned and converted to decimal, and the values put into arrays. (For about 20 seconds, it will look as though nothing is happening.) Your sprite will then be displayed as it would look in a program.

The program pauses again, and if you like the sprite, you can get a listing of the 63 decimal values that you can Poke into memory to display the sprite in a program of your own. Typing GOTO500: will put the list on the screen. GOTO700: will send it to the printer. Both lists are read across.

If you're not satisfied, and want to modify the sprite, do so. Then type GOTO200: to put the new values into the arrays and to display the revised version.

By typing GOTO600:, you can always have the computer redraw the picture for you. It will use the data in the arrays to do this. No matter what you

do to the drawing, the array data won't change until GOTO200: is typed. Of course, typing RUN will erase it. Table 1 summarizes the action of the GOTO commands.

Command	Action
GOTO200:	Scan the sprite drawing, convert it to decimal values and store them in an array. Display sprite.
GOTO500:	List the 63 decimal values on the screen. Read across the rows.
GOTO600:	Redraw the current sprite.
GOTO700:	List the 63 decimal values on the printer. Read across the rows.

Table 1. Summary of GOTO commands in sprite drawing program.

How the Program Works

This program works by the position of the drawing on the screen. If the screen should scroll up even one row, all the values for the sprite would be wrong. You must be careful to keep the cursor away from the bottom of the screen; that's why you should enter the GOTO commands about halfway from the top. The reason the GOTOs are followed by a colon is to keep the computer from trying to read the whole line, which includes part of your sprite drawing.

Line 45 dimensions the two arrays used and sets V equal to the start of the video display chip. Lines 100 through 720 call the subroutines that do the work of the program.

Subroutines 1000 and 1100 make the borders for the sprite drawing. Subroutine 1200 scans the area within the borders. If a 1 is found, it puts a 1 in the corresponding element of array AR. Otherwise, it puts a 0 in the array location. There are 504 ($3 \times 8 \times 21$) elements in array AR.

Subroutine 1300 takes the binary data from array AR, eight elements at a time, and treats this as an 8-bit binary number. This is converted to its decimal equivalent and is stored as one of the 63 words in array A1.

Subroutine 1400 displays the sprite. The 63 decimal numbers from array A1 are Poked into memory, starting at location 832. (Locations 828 to 1019 comprise the tape I/O buffer.) 832 is 64×13, so that with blocks of 64, this data is stored in the 13th block.

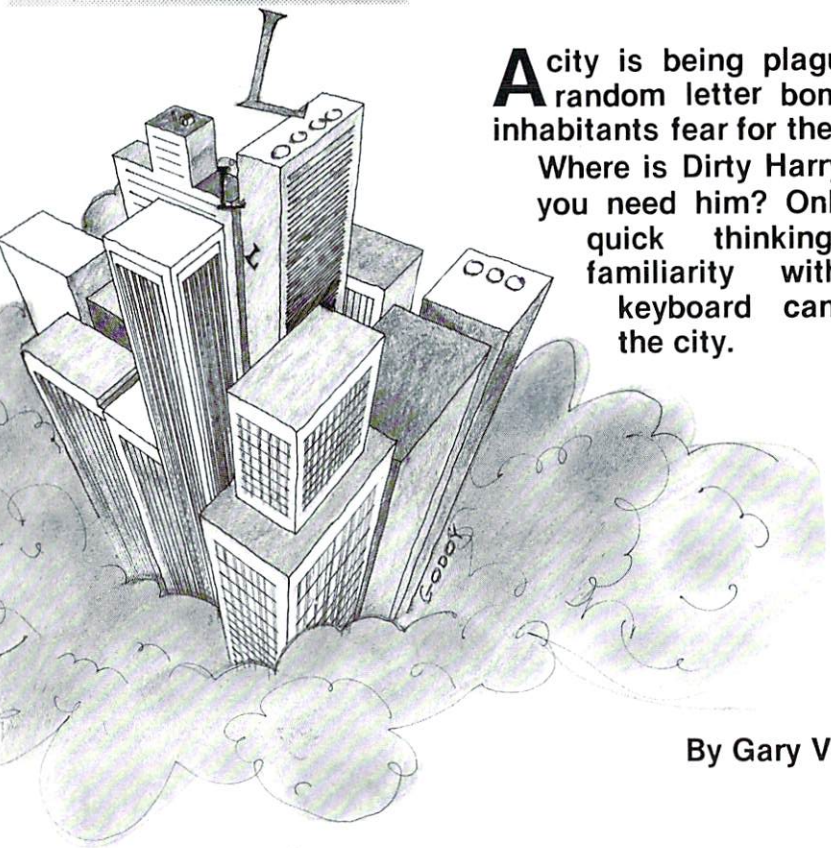
In line 1450, 2042 is the location that points to the data for sprite 2. The 13 is Poked into it because the data was put into the 13th block.

Location V + 21 enables (displays on the screen) a sprite. In this case, it turns on sprite 2 since a 4 (2×2) was Poked in. Line 1440 specifies the horizontal and vertical position of the sprite.

Subroutine 1500 lists the 63 decimal numbers that can be used to define a sprite in a program.

Subroutine 1600 takes the binary data from array AR and redraws the picture on the screen so it can be reviewed and modified if desired. R

MAD L BOMBER



A city is being plagued by random letter bombs; its inhabitants fear for their lives. Where is Dirty Harry when you need him? Only your quick thinking and familiarity with the keyboard can save the city.

By Gary V. Fields

Which is more important—learning on your home computer or having fun with it? That probably depends on whether you're a child or the child's parent. But Mad L Bomber is both fun and educational.

Play this game and you might learn to type, or at least become familiar with the keyboard. And as anyone in business knows, keyboards are everywhere.

RUN It Right
Commodore 64

Address author correspondence to Gary V. Fields, 86 Lanvale Ave., Asheville, NC 28806.

The Mad L Bomber attacks Anycity, U.S.A., with mean letter bombs. He can attack very fast if you want, or very slowly. Your skill with the keyboard should determine the speed.

The quiet calm of the title screen is shattered by a siren and attack L bombs (letter bombs). The sound on the program is captivating, and the better your monitor's or tv's speaker, the better the sound.

You are asked to pick a speed, and the fun begins. A city in low-lying fog is displayed. Suddenly the screen alerts you with a flashing "Code Red" alarm.

Out of the clouds comes the villain himself. But don't pay him too much attention—he's just trying to distract

you so his randomly dropped bomb can make it to ground zero.

Sometimes the letter bomb comes from beneath Mad L Bomber, but usually it drops from another part of the sky. You must recognize the letter and press the same letter on the keyboard before the bomb twists, turns and explodes on the city below.

If you are fast and correct, the bomb will self-destruct—tumble and explode. Then, the Mad L Bomber will beat a quick retreat into the clouds above.

But don't go away. He'll be back for a total of 21 bomb runs or until he wins, whichever comes first.

If you match the Mad L Bomber letter for letter, you'll win, and Anycity, U.S.A., will thank you with hearty applause. The program then gives you a chance to choose a speed and begin again.

3—Goes to title screen and siren.
4—Sets up for sprite. BX and BY are the sprite location variables.
5—Sets sound variables and Pokes Mad L Bomber yellow.
6—Requests speed and Pokes Bomber into upper left corner of screen.
7–15—Input the speed to use on bombs.
16—MC is the variable that counts bombs.
20–22—Poke the screen black and check to see if game is half over—MC>20.
23–25—This is the halfway-through subroutine. If MC>20, then program goes here.
30—TT keeps track of where bomb is on the screen.
31–36—Flash the Code Red warning with beeping sound.
37—Double checks input in main loop will begin (tab=N).
40—N chooses where bombs will start on screen (tab=N).
50—Pokes screen black and moves cursor to home position (CHR$(19)).
70–82—Define city and sky.
94—CR limits rising fog and clouds from going above screen.
95–98—Randomly Poke fog or clouds on screen.
100—LE chooses which letter will be a bomb.
110—Makes sure bomb is a letter and not a number or graphics symbol.
120—GETK$ starts looking for keyboard response to falling L bomb.
130—If proper response was made, this calls up explosion in routine 155–250.
155–250—Falling letter tumbles, explodes, and Mad Bomber heads for cover.
170–178—Make bomb appear to explode.
200—Pokes all the sound off.
250—Adds to MC counter. Pokes Sprite off and returns for another bomb.

300–350—The falling bomb sound.
310—Converts N (tab value) to value with an angle for dropping bomb's twist.
311–312—Keep the bomb within the screen's borders.
360—Detects when bomb touches city.
370—Erases each letter and leaves a red trail where it had been.
800–808—Make screen and city appear to explode.
810–820—Last screen.
820—Goes to hear the siren again.
830–854—Ask if player wants to continue or quit.
1050–1054—Explosion sound.
2000–2028—Title Screen.
2029—Reads Sprite data lines and Pokes it into the proper memory location.
2030–2080—Siren sound.
4000–4070—Winner routine.
6000–6003—Data lines which define Mad L Bomber Sprite.

Table. Line by line explanation of Mad L Bomber program.

But if you fail, the city explodes, you are notified of how many letter bombs you stopped, and you are given the chance to play again.

By the Way

Also note that an early morning fog rises during the first half of the attack. After the tenth bomb run by the Mad L Bomber, you are given a short rest. When the attack continues, the fog is gone, but now an ever-thickening cloud cover makes early recognition harder and success tougher. After each attack, the fog or clouds thicken and rise.

R

EASY MUSIC MAKER

Even if you're not a musician, you can make a maestro out of your C-64 with this magic music program.

By Gary V. Fields

Total Music 64 gets sound out of your Commodore 64 and into your ears. This program takes all the work out of adding sounds or songs to your own programs.

If you know nothing about the sound interface device (SID), Voice 1, 2 or 3, or the difference between C# and G, help is here.

If you couldn't care less about the items mentioned above and only want to get to the sound, then Total Music 64 was written for you.

But don't be surprised if you also end up learning a lot about SID; Total Music 64 lets you whistle while it works.

Type in the program as listed. (Be careful not to leave out any semicolons.) Save the program before you run it the first time.

Now run the program. The title page and some information about the program and how to use it will dance across the screen. Press the space bar to begin.

The next screen gives you a chance to select and define your own voice with:

RUN It Right

Commodore 64
11K required

Address author correspondence to Gary V. Fields, 86 Lanvale Ave., Asheville, NC 28806.

volume, waveform, attack/decay, sustain/release and duration of note. Press the return key and the program will default to preselected values that you can later experiment with, if you wish.

Next is the Practice screen. When you press A, the letter will appear on the screen, and you'll hear the musical note A (octave 4). When you press shift/A, you'll hear A# and both characters will appear on the screen. The same pattern holds true for A–G. All other keys are empty except P, the function keys (F1–F8), the INST/DEL key, the return key and the up-arrow key.

P is for pause; you won't hear any sound when it appears.

The up-arrow key erases the screen. (You can play with the whole screen display, but you should reserve the Practice screen for the area above the midline.)

The return key starts another line of practice notes.

INST/DEL erases each note, one at a time.

F3 exits the practice screen and puts you into Total Music. Everything on the screen will be erased except the notes in the practice area. A new menu, offering additional options, will be displayed.

F7 exits back to the screen where you select and define your own voice.

Play with the Practice screen for as long as you please. When you get a song or series of notes you like, press F3 and go to Print & Play.

Print & Play

This is where all the fun and real work is done. The screen should now display:

F1 DATA ONLY F2 TOTAL PRINT
F3 PRACTICE F4 PRINT NOTES
F5 PLAY TUNE F6 DURATION
F7 RESTART
P PAUSE (UP ARROW) ERASE ALL

The notes above the midline (from the Practice screen) should still be on the screen, and a little right-arrow symbol should race across the screen, erasing two lines below the midline.

You'll still hear the notes when you press them, but now they'll be displayed below the midline.

Your notes will be placed in an array, which was dimensioned in line 15 to be a maximum of 200 notes. The lower half of the screen can display about 200 notes. During this time, copy your practice notes or play something else.

If you want to start over, press the up-arrow key, and all the notes you've just played (except your notes in the practice area) will be erased. If you press F3 twice, you'll erase everything. If you press F3 once, you'll return to the Practice screen.

After you've arranged some notes on the screen into a tune, press F5, and your tune will be played as it would sound if it were in a program. (Note: C,C,C will sound the C note for three duration counts. If you want to hear the C note three distinct times, play C P C P C.)

When It Works, It Plays

When your tune sounds just right, press F2 (make sure your printer is ready). The program will print out a total program listing. To use it, all you need do is copy it into a program. Everything's there—every Poke, every Read, every For...Next loop and every piece of data.

However, you may have to make a change in copying the data. The program lists all the data on one line. If your tune requires more than one line of data, just add another data line number and continue.

F1 prints only the data needed for the tune. Later, you'll probably choose this most often.

F4 prints the actual notes/letters. (Total Music 64 uses octave 4.) By using these notes, you could look on page 152 of the *Commodore 64 User's Guide* and translate each note into high and low frequency for different octaves.

You're asked to name each tune that's printed. Then the printer takes over. When the printer finishes, it will also print out a total count of the data. You might need this if you're using other Read statements in a program.

F6 lets you choose another duration count for your tune without having to exit this mode. Note: INST/DEL was not included in Print & Play mode because the notes are placed in an array. If each note used only one character, then INST/DEL could have been included. But, for example, C# takes up two characters. Erasing would have thrown off the array count.

I hope you have fun with Total Music 64, and that it adds lots of sound to your programs.

Here are some tunes with which you can experiment:

Old MacDonald—F P F P F P C C P D P D P C C P A P A P G P G P F F

This Old Man—G E G P G E G P A P G P F P E P D P E P F P E F G P C P C P C P C P D E F G P G

After you become familiar with the program, try the same tunes using different waveforms, attack/decay, sustain/release and duration values.

R

Quick color and sound flash sequences really test your memory and reaction speed in this fun program. What a feeling!

By Zoltan Szepesi

The Repeat the Sequence program enables you to play three games that exercise and improve both your visual and auditory memory. Another game in this program is useful for checking your reaction speed. The program (Listing1) is written in Basic for the Commodore 64, but it could be modified for use on other computers. (The VIC-20 version of this program was published in *Microcomputing*, January 1983, p. 86.)

The basic idea of these games is the same as that of the SIMON (copyright 1979, Milton Bradley Co.). However, by using the computer with its display, a better communication between machine and player can be achieved.

There are four different color squares displayed at four different places on the TV screen. Single color flashes are presented in random order, each accompanied by their special sound flash. You have to repeat it by pressing the same color keys (without pressing the control key). The colors used are purple, green, blue and yellow: consequently, keys 5, 6, 7 and 8 have to be pressed. If you repeat the color and sound flashes correctly, the game continues. Descriptions of the four games follow.

Game 1. Create the Sequence.

After you have repeated the first signal your C-64 gave, you have to add another signal. Following that, you have to repeat the sequence of the previous signals and add another to it. Continue this way until a given number of steps are finished (see Table 1), when the C-64 salutes you with the first eight notes of Beethoven's 5th Symphony. If you were not fast enough, or if you made a mistake in repeating the sequence correctly, the computer gives a noisy sequence of the 5th Symphony and the game is finished.

Before starting with each game, you can choose one of four difficulty levels by pressing one of the programmable function keys (F1, F3, F5 or F7). Table 1 lists the different parameters defined by these keys.

When the function key F3 is pressed (after the C-64 asks for it at the start of the game), eight sequences have to be correctly repeated for successfully finishing the game. The time lag between signals will be short; you have to push the proper color within three seconds after the previous color was pushed.

RUN It Right

Commodore 64

Address author correspondence to Zoltan Szepesi, 2611 Saybrook Drive, Pittsburgh, PA 15235.

Game 2. Repeating Sequence.

The C-64 starts by giving one signal. After you have repeated it successfully, the computer repeats the previous signal and adds one new signal. You have to repeat this sequence again. In the following steps, your 64 repeats the previous sequence and adds a new one until the series is completed according to the number of sequences chosen.

Game 3. Changing Sequence.

This game is very much the same as Game 2. The only difference is that the computer does not repeat the previously given sequence, but always starts a new sequence with one more signal in it.

Game 4. Single Flashes.

In this game, the C-64 gives only one signal at a time and you have to repeat it within one second.

Table 2 shows the statement numbers and subjects of the different sections of the program. The list of main variables is shown in Table 3.

The variables N1, T1 and T2 are fixed by the four programmable keys in statements 55 to 70. You can change them by changing the numbers in these statements. The time between flashes (T1) is only a relative value. It is in addition to the time it takes to display the color square. T2 is in seconds. In Game

Function key	Number of sequences	Time between signals	Time allowed to you
F1	4	100	5 seconds
F3	8	50	3 seconds
F5	16	10	2 seconds
F7	32	1	1 second

Table 1. Difficulty levels within a given game.

Statement No.	Subject and remarks
5–10	Title and author
15–95	Initialization. Choose game number and difficulty level
200–230	Main program of Game 1
250–275	Main program of Game 2
300–310	Main program of Game 3
350–360	Main program of Game 4
400–435	Subroutine of color and sound flashes
450–485	Subroutine for repeating the sequence
500–520	Error messages
525–560	Music program
570–580	Correct finish. Playing 5th Symphony. (Data in 540)
600–640	Repeat last correct sequence?
650–680	Want to continue?
700–710	For stack clearing

Table 2. List of principal sections of the program.

4, T2 is redefined in statement 355.

Since from the subroutine "REPEAT THE SEQUENCE" (statements 450 to 485) the program exits in given circumstances without using the Return command, the stack could be filled (after about 13 games at the same game number) and an error message "OUT OF MEMORY" could turn up. To avoid this disaster, three more Return commands were put in this subroutine with flag Q. The subroutine "FOR STACK CLEARING" (statements 700 to 710) gives the proper Jump statement.

Similar stack filling can also happen when a For...Next loop is left before ending it. The first part of statement 515 clears up this problem.

The program needs 3568 bytes of memory; another 400 bytes are needed when it is executed. R

Variable	Remarks
B$	11 cursor down + purple code
C$	marking of color spots
C0	color memory location
G	number of games in the same kind of game
K(N)	position of color spot at *n*-th flash in the sequence
N1	maximum number of flashes in a sequence
NM	number of flashes in a given sequence
P	number of flashes in Game 4
Q	flag when exit from subroutine
S	screen memory location
SI	duration of musical notes
T	voice number address
T1	time between flashes (see explanation below)
T2	time delay allowed, in seconds, when repeating flashes
T3	clock status, when measuring time delay T2
TN	pitch code of musical notes
TT	total number of points in the same kind of game
X	game number

Table 3. List of main variables.

It's easy to keep your disk drive operations neat and tidy with this C-64 conversion of the DISK-O-VIC utility program that ran in *RUN*'s first issue.

By Cal Overhulser

In the premiere issue of *RUN* appeared a dynamite disk utility package for the VIC-20 called DISK-O-VIC by Thomas Henry. It is one of the most useful 1541 disk utilities I've seen, and it made disk drive housekeeping operations very easy on my VIC-20. I wanted the same capabilities on my C-64, so I decided to try converting DISK-O-VIC to DISK-O-64.

The main problem was that of converting the addresses for the system calls in DISK-O-VIC to those addresses appropriate for the C-64. The Kernal calls were easy, since they are the same for both machines and are published in several reference manuals. The real problem involved other system calls such as Warmstart, Reset and Printstring.

After some searching of the C-64 ROMs, I found the routines I needed. Table 1 lists the variable names from the original DISK-O-VIC assembly listing that require changes, along with their new system addresses for the C-64. Once I had the correct system addresses, the actual conversion became relatively easy.

First, I found the affected system calls every place they appeared in the original assembly listing and located their equivalents in the original hex dump. Next, I determined the changes necessary to fix the startup screen. I then loaded in DISK-O-VIC, made the necessary changes with a monitor and saved a copy of DISK-O-64 with the same length and same capabilities as DISK-O-VIC.

RUN It Right

Commodore 64
1541 disk drive
Machine-language monitor

Address all author correspondence to Cal Overhulser, 15 Nutting Road, Westford, MA 01886.

Entering the Program

You'll need a machine-language monitor to enter the DISK-O-64 program from the hex-dump listing. After loading and entering your monitor, you begin entering the program at address $0801 and continue through $0D2F. Then use the method appropriate for your monitor to save DISK-O-64. Using the C-64 monitor from Commodore, you'd type:

```
S "DISKO64",08,0801,0D2F
```

Make sure you use $0801 as the start address so you can later load it like a Basic program. You now have a copy of DISK-O-64 that can be loaded and saved like any Basic program. Then you exit the monitor and reset the C-64, either by typing SYS64738 in the Direct mode or by turning the power off and on.

Now load and run DISK-O-64 like any Basic program. Just type LOAD "DISKO64",8 to load it into your C-64, and then type RUN. If all goes well, the startup screen appears, and DISK-O-64 is now in place, protected in upper memory.

Table 3 lists the new commands now in place. If you read the original article on DISK-O-VIC, you'll see that all commands remain the same for DISK-O-64. Use a scratch disk and experiment with each command to become familiar with them (also to make sure everything is working OK, with no typos).

DLoad/DSave Restriction Changes

I found one minor irritant in the original DISK-O-VIC. I couldn't use DLoad/DSave on a hybrid program (one containing both Basic and machine language). It would appear to load and save all right, but I noticed that the saved program had fewer blocks than the original.

The real problem was that I didn't realize what was happening until one day a favorite hybrid program (DISK-O-VIC) wouldn't run, and I had to type the whole thing in again. In all fairness to Mr. Henry, I must say he mentioned this restriction in his article, but in my haste I failed to note it.

I traced the problem to the DLoad routine in DISK-O-VIC. DISK-O-64 has this modified so DLoad/DSave can be used with hybrid programs as long as they are loaded like a normal Basic program, i.e., LOAD"NAME",8.

The only exception is with the Append command, which can be used to append only pure Basic programs, not hybrids. Practically speaking, the need to append the Basic portions of hybrid programs is extremely rare (I've never done it).

The changes to edit the original DISK-O-VIC are shown in Table 2 for the convenience of DISK-O-VIC users who may want to modify their copies.

R

Label	VIC	C-64
WARMST	C474	E386
WAIT	C48C	A48C
INFIN	C49F	A49F
CHAIN	C533	A533
CLR	C659	A659
INTEGR	C96B	A96B
PSTRNG	CB1E	AB1E
ERROR	CF08	AF08
PRLINE	DDCD	BDCD
CHROUT	E742	E716
RESET	FD22	FCE2

Table 1. DISK-O-64 label equate changes from DISK-O-VIC assembly listing.

VIC Hex Address	New Hex Value
126E	86
126F	2D
1270	84
1271	2E
1272–127B	EA
127C	20
127D	33
127E	C5

Table 2. DISK-O-VIC changes for DLoad/DSave with hybrid programs.

Append—This command allows a Basic program to be appended from the disk to a program in memory that has lower line numbers. The proper syntax is: APPEND"NAME".

Catalog—Typing CATALOG will read the directory from the disk and display it on the screen without destroying the program in memory.

Collect—This performs a validate, which means it tidies up the disk and makes all unused blocks available.

DLoad—Acts like the normal Load command, but you don't have to type ,8. It also initializes before, and checks for errors after, it loads. Both Basic and hybrid programs can be loaded as long as the hybrid programs are normally loaded like all-Basic programs.

DSave—Just like DLoad, but saves programs to the disk.

Header—This command will format a disk. Since all data will be destroyed, it asks "ARE YOU SURE? Y or N." The correct syntax is: HEADER"NEWNAME",Ixx. You must use the ,I. The xx is any ID you want to assign (different for every disk you own).

INIT—This is the same as OPEN15,8,15,"I":CLOSE15 in Basic.

Kill—This does a reset of the C-64 much like turning the power off and on or typing SYS64738.

Off—This one will disable DISK-O-64, but leave it and any other program in memory intact. DISK-O-64 slows down Basic a little, so you can turn it off when you're interested in maximum speed. To turn it back on, type: SYS256*PEEK(56)+PEEK(55).

Rename—Allows you to rename a program that already exists on the disk. The syntax is: RENAME"OLDNAME"TO"NEWNAME".

Scratch—This will scratch a program on the disk; it is equivalent to OPEN15,8,15,"S0:NAME":CLOSE15 in Basic. It also asks "ARE YOU SURE? Y or N." The correct syntax is: SCRATCH"NAME".

Send—With this one, you can send any command to the disk that you can send in Basic; it is the same as OPEN15,8,15,"xxxx":CLOSE15 in Basic, where xxxx is the command string. The proper syntax is: SEND"xxxx".

Status—Displays the disk status without executing a program. When you get a disk error, just type STATUS.

Table 3. Explanation of DISK-O-64 commands.

DISK-O-VIC

Catch Saturday Night Fever with this utility program that gives you 13 disk-related commands and will keep your VIC-20 and 1541 disk drive dancing. Move over, John Travolta.

By Thomas Henry

The Commodore VIC-20 computer and 1541 disk drive make a very powerful computing combination. The VIC-20, of course, is a full-fledged 6502-based computer, offering many professional features such as a thorough set of Basic commands, a professional keyboard and expandable memory options.

A beginning system often starts with a cassette unit for mass storage, but as the user's level of expertise rises and the need for faster I/O becomes more important, a disk drive becomes essential. The 1541 drive, like all Commodore's floppy disk units, is intelligent. This means it is a computer in its own right and is able to perform many functions with the intervention of the host computer.

RUN It Right

Editor/assembler or
machine language monitor program
VIC-20
1514 Disk Drive

Address author correspondence to Thomas Henry, Transonic Laboratories, 249 Norton St., Mankato, MN 56001.

In fact, the 1541 contains its own 6502 microprocessor, a couple of VIAs (versatile interface adapters), 2K of RAM and a complete operating system in ROM. This leads to two important facts. First, since the 1541's system is so complete, it steals no user program RAM from the host computer. Unlike many disk drive/computer combinations, a VIC-20 has just as much program space with disk drive as without it.

Second, since the 1541 is intelligent, you can externally program it to perform many useful functions. The unit is essentially open-ended in the sense that if a particular function doesn't already exist in the disk operating system, you may write a program to generate such a function.

DISK-O-VIC

This article describes a utility package, called DISK-O-VIC, which adds thirteen new disk-related commands to the VIC-20. These commands become part of Basic and you may use them in the immediate mode to simplify disk drive housekeeping operations. Some of the commands, such as DLOAD and DSAVE, are extensions of old Basic commands. Others, like Scratch and Rename, are for keeping disks neat and orderly. Finally, another group adds features such as error message readout, directory display and so on.

The DISK-O-VIC utility package is written in machine language for maximum speed and flexibility. After it has been loaded and initialized, it may be left in place for an entire programming session. Due to the special loader feature (described later), DISK-O-VIC will sit at the top of memory and be free from Basic program interference. Thus, it adds thirteen new commands while

remaining transparent to the normal operating system.

The special loader feature also makes it possible to use this package in a VIC-20 with any amount of extra memory. It will not become obsolete if you decide to add extra memory at a later date. After installation and initialization, DISK-O-VIC consumes 980 bytes and leaves zero page intact.

Thirteen New Disk Commands

Before describing DISK-O-VIC's mode of operation, I'll examine the new commands so you can see just what they do. (For full details, see the accompanying table of commands.)

Whenever a floppy disk is inserted into the 1541 and is subsequently accessed, a special chart, called the block availability map, is created in the drive's memory. This chart contains special information about the disk currently in the drive, such as how the disk has been partitioned, what blocks are free and other various allocation matters. Fortunately, the disk drive keeps track of this somewhat esoteric information, so you rarely need to be concerned with it.

The process of creating this chart is called initialization. You must initialize a disk if it is to be properly written to or read from. (Note that some non-Commodore disk drive systems use the term initialization to mean "format the disk," a process which can write over or destroy data. This is not the case with the 1541 disk drive.)

To ensure that the information in the drive's memory is up to date, you should initialize the disk often during a session. The 1541, as it comes from the factory, will generally perform self-initialization during the execution of various commands. To add a margin of safety, an automatic initialization precedes every command in DISK-O-VIC. Though this may be somewhat redundant, the process takes only a second and goes a long way toward reducing problems. It never hurts to over-initialize!

The DLOAD and DSAVE commands work exactly like the VIC-20's Load and Save commands, except that the computer knows automatically that the proper device to access is the disk drive (device number eight). These commands are for Basic programs only. Do not try to DLOAD or DSAVE machine language or hybrid programs, for the commands make certain assumptions about the start of program space that may or may not be true for machine language programs. In general, all the commands in DISK-O-VIC assume you are working in Basic.

DLOAD and DSAVE automatically check the error channel after an operation to see that all went well. If an error is detected (Drive Not Ready, File Exists, File Not Found, etc.), the message is printed to the screen and the file is closed down.

Catalog is an interesting command. Unlike the old way of doing things, you may print the disk directory or catalog directly to the screen, thus preserving any programs in memory. To stop the listing to the screen, simply push the space bar once; to resume the listing, push the space bar again.

The purpose of the rest of the commands should be obvious. Just look over Table 1 and perhaps refer to the 1541 disk drive manual from time to time. Users of larger (and more expensive) Commodore computers will probably recognize many of the commands. Unlike the VIC-20, computers such as the PET and SUPERPET already have a set of disk commands very similar to those provided by DISK-O-VIC.

How the Program Works

Now that DISK-O-VIC has been

introduced, we'll look at how it works. As an aid to understanding, Listing 1 presents an assembler listing for the complete program. Since assemblers are starting to become more common for the VIC-20, you may wish to enter the source code and assemble your own version. But the assembler listing has been provided for its educational value, and most users will want to enter the object code directly. A hexdump of this code is provided in Listing 2.

Most problems with detail can probably be cleared up by studying the comments in the listing. As an aid to understanding, however, I'll describe the basic structure of the program. To do this, some consideration must be given to the way Basic fetches and executes a command.

When interpretative Basic is in action, a pointer must seek commands by parsing or scanning the input line. The interpreter checks the input line, character by character, in hopes of finding a command that it recognizes. Thus, if you want to add new commands to Basic, you must put a "wedge" into the parser routine, diverting attention from the normal scanning procedure to a new one. Essentially, the parser is forced to look for the new commands first. If it can't match a command with any on the new list, then control is sent back to the normal system and it will check the input command against its old list.

The first block of code in Listing 1, lines 00072–00080, is the initialization routine. This code inserts a wedge into the normal parser routine, so initialization need occur only at the start of the session. After initialization, the Basic parser will always check first for DISK-O-VIC commands.

The next block of code occurs in lines 00086–00165. This is the parser add-on. As mentioned above, the parser will be directed to this routine each time a command is input to the computer. The key instruction in this block occurs at line 00100. The stack is examined for any "RTS" (return addresses). If the address on the top of the stack indicates that the parser has come from the VIC-20's "waiting for a command" state, then action is taken. If some other address is found, then the parser is allowed to continue its normal activity.

Assuming that the test has been passed and the VIC-20 is indeed waiting for a command, the input line is then checked character by character. This occurs in the block of code labeled "Parser Routine," lines 00119–00150. The input is checked against the list of DISK-O-VIC commands held in a table at lines 00573–00586. If a match is found, then an "action address" is formed, and control is passed to the proper subroutine.

Subroutines

The great bulk of the program is devoted to the various command subroutines. To make them easier to find, these subroutines have been arranged in alphabetical order, with Append coming first, then Catalog, and so on. Although at this point the program may look complex, it is actually quite easy to analyze if you attack one small function at a time.

Toward the end of the program, at line number 00452, some general purpose subroutines are presented. These are commonly used by the rest of the program to fetch file names, get disk parameters, print messages to the screen and so on. In general, they have been assigned labels or names that relate to the functions they perform.

DISK-O-VIC ends with various data and address tables. First is the table of keywords, described above. Then follows a table containing the addresses of

the command subroutines. Finally, a set of variables is created for the purpose of saving registers and so forth. By assigning variables to this area, use of critical zero page locations is avoided.

Since details have a way of clouding the issue, here is a summary of the overall structure just described:

- Initialization
- The "wedge" into the parser
- The new parser routine
- The command subroutines
- General purpose subroutines
- Data, addresses and variables

Before leaving the theoretical aspects of DISK-O-VIC, we should look at the table of equates in the assembler listing. The table uses about a dozen zero-page locations, but since they are used for their normal purposes, the operating system takes little heed. Actually, locations $FB through $FE are free zero-page locations and are not used by the VIC-20 at all.

Since this is a disk operating system, it is assumed that the cassette tape unit won't be used. This frees a large block of space, starting at $033C, that is normally employed as a cassette buffer. This block can therefore be used for the disk command buffer. Even if a cassette unit is tied to the VIC-20 along with a floppy disk, no conflict should occur, since both units would rarely be accessed simultaneously.

ROM Routines

Next in the table of equates is a large group of subroutines contained in the VIC-20 operating system's ROM set. Even if you have no use for DISK-O-VIC, this table should prove to be helpful. By using standard ROM routines like these in homebrew programs, it is possible to save many hundreds, even thousands, of bytes. Let's examine a few of the routines in greater detail.

The VIC-20 ROM routines fall into two rough categories. The first group, called the Kernal, is composed of major routines for input and output operations. (For some unknown reason, "Kernal" is the spelling officially recognized by Commodore.) These Kernal routines are special in that several models of Commodore computers have the same routines occurring at identical addresses.

For example, the "output a byte" routine occurs at $FFD2 for *all* makes of Commodore computers. In general, however, the Kernal routines are all identical only for the VIC-20 and Commodore 64. This implies that software making extensive use of the Kernal should be simple to transfer from the VIC-20 to the Commodore 64, and vice versa.

The Basic commands are another category of routines. Their locations will vary from machine to machine, but are likely to be similar for all the Commodore computers. The routine at $CB1E, labeled PSTRNG in listing 1, is one of these. When called, it will print to the screen a string aimed at by the accumulator and y register. It will keep printing the string of characters until a terminating zero is detected.

To learn more about the operation of the VIC-20, be sure to carefully examine the table of equates in Listing 1. In most cases, the locations and routines have been given meaningful labels to aid interpretation, and the comments will fill in some of the details. For more information on the Kernal routines, refer to the *VIC-20 Programmer's Reference Guide* (Howard W. Sams and Co., PO Box 7092, Indianapolis, IN 46206). To understand the non-Kernal routines, you will need a more extensive memory map.

Entering the Program

Having covered the theory and operation of DISK-O-VIC, we must consider the practical side of things. You should create a disk copy of the object code so that the utility is always handy. To this end, the hexdump in Listing 2 corresponds to the source code in Listing 1. To use it, you enter the hexadecimal numbers into the VIC-20, then save it to disk. Thus, whenever you want to invoke DISK-O-VIC, you have only to load the code and initialize it.

Since the program is in machine language, you will need a machine language monitor to enter it. The VIC-20 has no resident monitor, but add-on monitors are starting to appear with increasing frequency. Two good choices are VICmon or Tinymon.

VICmon, made by Commodore, is the official machine language monitor for the VIC-20 and offers many commands. It comes in cartridge (ROM) form, and simply slips into the expansion port. Tinymon, on the other hand, is a tape or disk-loaded monitor. The advantage of Tinymon is that you can punch it in yourself and save quite a bit of money. It doesn't support as many commands as VICmon, but that doesn't matter for the purpose at hand. All you need are the S (save) and M (memory dump) commands. Hence, either monitor will do.

(For a full discussion of Tinymon, see Jim Butterfield's article "Tinymon1: A Simple Monitor for the VIC," in the January 1982 issue of *COMPUTE!*, p. 176.)

To make a copy of DISK-O-VIC for your computer, follow these instructions carefully:

- Disconnect any memory add-ons. DISK-O-VIC must be entered on a stock machine.
- Load in a machine language monitor. Either tape/disk-based or cartridge monitors will do.
- Using Listing 2 as a guide, punch in the object code. You will start entering code at location $1000 and continue upward.
- After you finish entering the code, modify the following locations. Put the data byte $2F into locations $2D, $2F and $31. Put the data byte $15 into locations $2E, $30 and $32. These are all zero-page locations.
- Exit the monitor to Basic with the X command.
- Now save the program using the ordinary VIC-20 SAVE command. You may save the program to either tape or disk.
- If you wish, reconnect any memory add-ons that you have.

You now have a full version of DISK-O-VIC ready to go. The code just entered and saved is very special. You can load and run it just like any Basic program. When you run the program, a special loader automatically relocates DISK-O-VIC to the top of memory, wherever that might be. Also, the loader instantly compensates for any extra memory that might be attached to the VIC-20.

Keypunching this program can be very tedious, so try to share the task with other users. One consolation is that even though the program is in machine language, it looks like Basic to the VIC-20. This means that you can make backup copies quite easily. To do so, simply load DISK-O-VIC (don't run it) and save some more copies by using the ordinary Save command.

Conclusion

The practical value of DISK-O-VIC should be obvious, but the program should also serve as an example of how a complete disk operating system can be

implemented on the VIC-20. The computer clearly contains many powerful routines in ROM, and it behooves every user to learn as much as possible about them. The program also shows that the 1541 disk drive is an extremely flexible unit.

Programming the VIC-20 and 1541 in machine language to perform new and exotic commands is not as difficult as it may at first seem. The key, of course, is to break the problem down into a series of smaller subroutines, making as much use as possible of the various ROM routines available. This was the very procedure used in DISK-O-VIC. R

Table 1. Explanation of DISK-O-VIC commands.

Append. This command allows a program from disk to be appended onto another in memory. To keep things simple for the VIC-20, it is important that the program in memory have line numbers less than the disk program to be appended. The availability of this command makes it possible to build large subroutine libraries from which complete programs may be assembled. The proper syntax is:

APPEND "title of program" [return]

As with normal VIC-20 Basic, some abbreviations are possible. For example, instead of typing in the whole word APPEND, you may type "A shift-P." (All of DISK-O-VIC's commands may be abbreviated in this fashion. Just type the first letter of the command, followed by the second letter shifted.)

Catalog. To determine what is on the disk currently in the drive unit, type CATALOG and hit the return key. A directory listing will be printed to the screen so you can see all of the programs available. Note that unlike the method for looking at the directory normally employed by the 1541, Catalog will not disturb the program sitting in the VIC-20's memory.

As an added convenience, a special pause feature has been added. Push the space bar once to pause the listing. Push it again to resume. You may also hit the run/stop key to terminate a listing.

Collect. Type this command and hit the return key, and the disk in the drive will be validated or collected. In simple terms, this will cause the 1541 drive unit to trace through the entire disk, making sure that all of the blocks are properly "connected." Any blocks that have been improperly allocated will be cleaned up and made available for more storage. The entire operation of this command is fairly complex, but basically it simply looks over the disk and tidies it up. Like initialization, it never hurts to use the Collect command often.

DLOAD. Acts just like the normal Load command but defaults to the disk drive automatically. For example, type DLOAD "program name" and hit the return key. The drive is automatically initialized, the program loaded and disk errors checked for. Just to put this into perspective, DLOAD is equivalent to the following steps:

```
OPEN 1,8,15,"I"
LOAD "program name",8
INPUT#1 disk error message, etc.
CLOSE 1
```

It is clear that DLOAD, though a simple command, does quite a lot. Incidentally, DLOAD may only be used for Basic programs or machine language programs that "look" like Basic. This limitation is due to the fact that the VIC-20 has a strange "sliding memory" loading format.

DSAVE. This is just like DLOAD, but saves a Basic program to the disk. The same initialization and error detection take place.

Header. This is a special command that takes a virgin disk and formats it for later use. Magnetic marks, which serve as guides to the 1541, are imprinted on the disk, and a title and identification code are assigned to it. The

syntax for its use is:

HEADER "disk name",Ixx [return]

where "disk name" is the name to be assigned to the disk. "xx" has been used here as the identification code. However, any two-character combination may be used. Note that the comma is necessary, as is the letter "I." Before the disk is headered, the query "Are You Sure? (Y/N)" is printed to the screen. An answer of "Y" will start the command; any other response will abort the process. Since the Header command overwrites the disk, it's important to provide this "Are You Sure?" feature.

INIT. As mentioned previously, every DISK-O-VIC command has automatic initialization built in. However, there may be times when a disk is acting troublesome and it is desired to force an initialization. To do so, simply type INIT, hit the return key and the disk will be initialized. This command is equivalent to typing

OPEN 1,8,15,"I"
CLOSE 1

Kill. This is a self-destruct command. When you have had enough of DISK-O-VIC for a programming session and wish to remove it from the computer entirely, type KILL and hit the return key. The computer will go through an entire reset, acting as though you had shut it off and then turned it on again. Do not confuse this command with Off (see below). Kill completely resets the computer. In general, use this command only when you wish to cause a cold start.

Off. This command turns off DISK-O-VIC, but leaves it in memory, safe and protected. Thus it may be returned to whenever desired. Since DISK-O-VIC slows Basic down somewhat, you may wish to turn it off whenever you're running a program to attain maximum speed. To turn it back on, simply type

SYS 256*PEEK(56) + PEEK(55) [return]

Rename. This command will rename a program on disk, without affecting any program already in memory. For example,

RENAME "old name" TO "new name" [return]

will change the name of the program to "new name." There are several things to note. The old name comes first, then the new name. The word "TO" must be present between the two names for the command to work. Finally, error detection is provided, so that it is impossible to Rename a file to a name currently in use.

Scratch. This command lets you scratch a file or program from the disk. Simply type SCRATCH followed by the name of the file, and hit the return key. Once again, the query "Are You Sure? (Y/N)" is presented. A response of "Y" will cause the file to be scratched.

Send. This is a general purpose command and can be used to send some of the standard Commodore disk commands to the drive unit. For example,

SEND "I" [return]

will send the letter I to the disk, and thus cause an initialization. (Of course, DISK-O-VIC's command, INIT, will do the same thing.) As another example,

SEND "R:new name = old name" [return]

will cause the file "old name" to be renamed. Since other commands in DISK-O-VIC cover most contingencies, the Send command is probably not needed often. But it's nice to have it handy for advanced disk programming operations. For the record, Send is equivalent to

OPEN 1,8,15,"command"
CLOSE 1

Status. This is a troubleshooting command that allows you to chase down the cause of a disk operation failure. If the red error light on your disk drive comes on, type STATUS and hit the return key. The light will go off, and an error message will be printed to the screen. This message will describe the error and where on the disk (in terms of track and sector) the problem was encountered. If everything is OK, no message is printed. To test this command, type the following:

OPEN 1,8,1,"GARBAGE" [return]

The 1541 drive will whir, and assuming that there isn't a file named "GARBAGE" on the disk, the error light should come on. Type STATUS, and the error message will be printed to the screen. Refer to the 1541 disk drive manual for a full explanation of the error messages.

You want to put more creativity into your programming, but your VIC-20's character set just doesn't satisfy your needs. Your only solution is to generate your own characters. Here are some valuable tips to help you design custom characters for your games and graphics with speed and ease.

By Stephen Erwin

If you're like most programmers interested in games or graphics, you sooner or later reach a point where the standard VIC-20 character set no longer satisfies your need for creativity. Although there are many interesting characters to choose from, your best solution when a game calls for spaceships or funny little men is to design a custom-made set of programmable characters.

Memory Moves

The basic techniques are fairly simple, but they do require a bit of background information before they can be understood. For starters, VIC-20 character memory is stored in ROM, which cannot be changed. Characters can be changed only when they are stored in the user RAM. Therefore, in order to create any new characters, the VIC character memory must first be moved into the limited locations in RAM that the VIC-20 video chip can access.

RUN It Right
VIC-20

Address author correspondence to Stephen Erwin, 102 Hickory Court, Portland, IN 47371.

The standard locations on the unexpanded or 3K expanded VIC are at the top of user memory in 7168, 6144 or 5120. Location 7168 will store 64 characters; location 5120 will store all 255 characters. You move the character location by Poking location 36869 with the proper code. (See Table 1.)

When you choose a character location, it's important to remember that you must subtract the memory used by the character set from the RAM available for programming. While location 7168 uses only 512 bytes, location 5120, which allows 255 characters, uses 2560 bytes, leaving only 1024 bytes for the rest of the program. For this reason, it's important to use no more characters than you absolutely need.

Another important consideration is that the VIC stores some types of variables at the top of user memory. To protect your character set from these variables, you must Poke locations 52 and 56 with the proper code. Table 1 shows the codes for moving a character set and protecting it.

Try entering POKE56,28:POKE52,28:POKE36869,255. The screen should now be filled with junk. This is because, although you've moved and protected the character location, you haven't yet put any characters in it. To return the screen to normal, POKE36869,240.

The following routine will Peek the standard character location and move 64 characters to the new location at 7168.

```
10 POKE56,28:POKE52,28:CLR
20 FOR T=7168 TO 7679:
```

```
    POKE T,PEEK(T + (32768 – 7168)):NEXT
30  POKE36869,255
```

When this is entered, the only noticeable change is that the cursor disappears. This is because the screen Poke number of the reverse space that the cursor uses is 160, and the new character set contains only 64 characters.

To adjust this formula to move more characters, change the codes in lines 10 and 30 to the proper codes for the new location and substitute the new location for 7168 in line 20.

It's also possible to move individual characters into the new character set. Use the following formula, where X equals the screen Poke code of the character in ROM, and Y equals the screen Poke code of the character to be replaced.

```
FOR T=0 TO 7:POKE7168 + Y*8 + T,PEEK
(32768 + X*8 + T):NEXT
```

For example, if you enter the following, hitting the X key will print a ?.

```
FOR T=0 TO 7: POKE 7168 + 24*8 + T,PEEK
(32768 + 63*8 + T):NEXT
```

POKE 36869	LOCATION	POKE 52 AND 56
240	ROM MEMORY	
253	5120	20
254	6144	24
255	7168	28

Table 1. Codes for moving and protecting a character set.

Design Originals

You're now finally ready to begin designing custom characters. Each one is made of 64 small dots on the screen. It takes eight bytes of memory to store one character, with each byte made up of eight on-off switches called bits. If the bit is turned on, so is the corresponding dot on the screen.

The eight bits within each byte are assigned the following values, which are the powers of 2 up to the seventh power: 128,64,32,16,8,4,2,1. Using (bit on) or not using (bit off) these numbers in all possible combinations gives you all byte values from 0–255. Fig. 1. shows the bit structure of a character resembling the profile of the space shuttle.

The numbers on the right in Fig. 1 represent the values obtained by adding together the values of the individual bits in each byte. To replace the @ with this character, simply Poke the above values into the first eight locations of your RAM character memory.

The standard method for doing this uses data statements as follows:

```
 10  READ A:IF A = – 1 THEN 100
 20  FOR T = 0 TO 7 :READ B : POKE(A*8) +
     7168 + T,B: NEXT
 30  DATA 0, 0, 0, 128, 192, 254, 255, 0, 0, – 1
100  END
```

The first data number is the screen Poke code of the character being replaced. The – 1 tells the program that the last character has been entered. If more than one character is entered, the – 1 is used only after the last character's data line.

An even easier way to make custom

characters is with the programmable character generator. When the program is run, it moves 64 characters into user RAM, pokes in any new characters that have been designed and then stops to let you test the new characters. Entering CONT places the character generator itself on the screen.

0	0	0	0	0	0	0	0	=	0
0	0	0	0	0	0	0	0	=	0
1	0	0	0	0	0	0	0	=	128
1	1	0	0	0	0	0	0	=	192
1	1	1	1	1	1	1	0	=	254
1	1	1	1	1	1	1	1	=	255
0	0	0	0	0	0	0	0	=	0
0	0	0	0	0	0	0	0	=	0

Fig. 1. The bit structure of a character resembling the space shuttle.

The VIC will draw a box with the numbers of the bytes on the sides and the numbers of the bits at the top and bottom. The bits are numbered from seven to zero to show the power of two that represents the value of that bit. For instance, $2^7 = 128$, the value of this highest bit.

Draw your new character by moving the cursor with W = up, X = down, A = left, D = right. Pressing the space bar will place a colored box under the location of the cursor. If an error is made, press F3. This allows you to erase the colored boxes by pressing the space bar. Pressing F1 returns the program to the drawing mode.

When the character is finished, move the cursor below the box and press F5. The program will then print out the Poke values for the new character and ask for the screen Poke code of the character to be replaced. See the user's guide on page 141 of the manual. Remember also that the character set uses only characters from 0–64.

When you enter this number, the program automatically writes a data line for the new character and adds this line to itself. It next returns to the beginning to enter the character into the character set and then stops so you can test the character by typing the key of the character that was replaced.

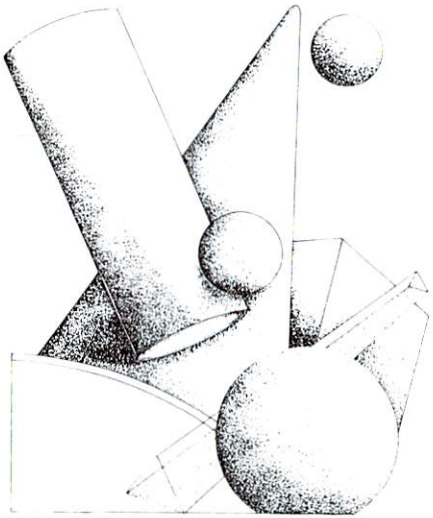
At this point, never use the return key except to continue the program by entering CONT or GOTO8000. If you do not like a character, simply make a new character and re-enter the same screen code as before. Because it will have a higher line number, it will replace the first character.

When you have made all characters desired, enter GOTO9000. This will automatically delete the generator part of the program, leaving only a program for loading the new character set.

You can save this program to tape and load it just like any other. You can add games above line 200, or you can separately load the character set and a game that uses it. All you have to do is draw the characters; the generator does the rest of the work.

R

Doodle On Your VIC



Let your artistic fancy fly free with this program that turns your VIC-20 into an easy, clever and powerful doodler.

by Terence Bryner

How would you like to see your flights of fancy in high-resolution, color graphics on your VIC, without a lot of planning or bother? If you have 3, 8 or 16K memory expansion, you can doodle to your heart's content with this program, which runs with keyboard or joystick; use a printer, too, to preserve your finest efforts for posterity.

RUN It Right

VIC-20
3K, 8K or 16K expansion
printer desirable
joystick optional

Address author correspondence to Terence Bryner, 15 Crane Road, Groton, CT 06340.

To use the program, type in Listing 1 and save it. Table 1 is a summary of directions. If your VIC has only the 3K memory expansion, simply load and run the program; the operating system will start it at 1024, and the program protects the high-resolution graphics screen.

If you have more memory, type in the command line at the top of Table 1 *before* loading the program. This causes the operating system to load it beginning at 8192, above high-resolution screen memory.

The program first asks whether you prefer keyboard or joystick control. After you hit J or K, the display goes mushy while the screen is reconfigured to 20 characters by 22 lines, and a flashing black dot appears in a white screen with a black border.

The border color is a key to the doodling mode—black is Draw. If you manipulate the joystick, or press a movement key (see Table 1), a dot will appear on the screen. If you hold the joystick in one direction or repeatedly press the movement keys, you'll leave a trail of black dots.

The left-arrow key, or the joystick's fire button, shifts you to the Erase mode, where the border is white and the trail of dots, becoming one with the background color, are invisible; use this mode to correct mistakes. If you hit the left-arrow key or the fire button again, you'll return to Draw mode. And that's how you doodle.

Several enhancements are provided. Press the f1 key and your flashing dot disappears. Press it again and it reappears, red. You can change it to five other colors and back to black (you'll see that the first color was really white).

The background color can be changed to any one of sixteen by hitting the f3 key. The f6 key stops the program and returns the screen to normal.

The f5 key causes the program to enter or leave Text mode. In this mode, the border is yellow and a 20-character banner (initially blank) moves across the top of the screen. If you strike a key while you're in this mode, the letter appears in the upper right-hand corner, pushing everything else to the left; you can use this to title your masterpieces. The f6 key still stops the program, and f5 returns you to the Draw/Erase mode.

If you have a 1515 or 1525 printer, you can save your creations by hitting f2 for a large (8-inches high) picture, or f4 for a smaller one.

A Compact Program

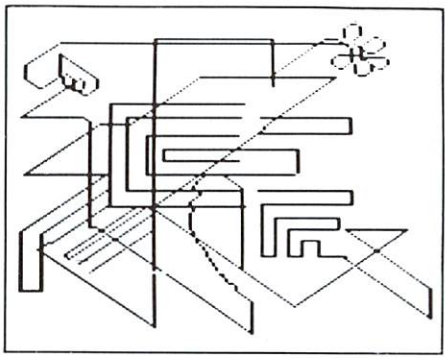
The program is short, in order to fit any memory expansion. Screen memory starts at 7680 for any memory configuration. The high-resolution graphics characters start at 4096 and continue to 7615 (this is why an expansion is necessary—there's no room left for the program).

Table 2 is a listing, by lines, of what each part of the program does. A few comments on specific techniques and variables may assist your understanding.

The screen is set up in 11 rows of 20 double-height characters. Screen memory contains numbers 0 thru 219, and character memory is set to start at 4096. This, along with clearing the screen and initializing variables, is done in lines 65–68.

Now a change to the bits in high-resolution character memory is reflected on the screen. X and Y contain the present position of the pen (flashing dot). BY is the address of the word in memory that contains that point, and BI is a pointer to the individual bit. CO is the character position used to set pen color on the screen. The array B contains the true value of the word at BY, a copy of the word with bit BI turned off and another copy with bit BI turned on. The point of interest flashes as the program, while waiting for input, cycles through the different values in B.

ED% is the variable that determines the Draw or Erase mode; C8 is the border-color variable, and contains the opposite of ED% (except in Text mode). C5 is the screen (background) color. C0 is the pen color—be aware



that the color control on the VIC is done in blocks of 8 × 16 dots, and the whole block changes color at once. The program spends most of its time in the loop from 25–30, waiting for input and flashing the dot.

One technique I've used extensively is multiplying by logical expressions. An expression, such as Y>0, has the mathematical value −1 if it's true, 0 if false. So line 31 places the value "one less than the present value of Y" into Y, if the present value of Y is greater than 0 (the preceding minus sign cancels the −1 of a true expression). If Y is zero, then the expression is false and its value 0; this effectively places a lower limit of zero on Y.

The following line:

```
Y=Y-1:IFY<0THENY=0:GOTO39
```

does *not* do the same thing. The GOTO at the end is only executed if the value of Y is initially less than one. Much more compact code can be written using this method.

Screen Printing

In printing the screen, I used an unusual technique, too. It is fairly direct for small copies obtained with the f4 key.

Lines 95–97 open a print file, set the Graphics mode and step through each line. Variable I1 contains the value of the first dot in each line, from the top of a 0–175 screen. A3 is a variable that gives the height of the character minus one. Characters are seven dots high except for the bottom line, and for small pictures, LS% is zero, so line 96 is not yet significant.

Lines 93–94 step across each line and print the variable Y$. The last character in the line is again only partial, so line 94 uses A5 to limit the subroutine to the screen and pad out the character with two blanks (CHR$(128)).

Lines 86–88 build the array H for one column of the character— it contains 0 if the bit is off, 1 if it's on. The 6 × 7 characters that the graphics printer produces cross character boundaries in the 8 × 16 screen, so you must go through the computations in lines 87 and 88 each time you build the array, which slows down the program.

In line 89, the dots are summed, the mandatory 128 is added and a graphics character is added to Y$, which represents one column of the total 6 × 7 character. That is a workable, if not optimal, solution to drawing the high-resolution screen.

However, a picture 176 dots high (on a printer that prints about 63 dots to the

Before loading with 8K or larger memory expansion, type in:

POKE 44,32: POKE 642,32: POKE 8192,0: NEW{RETURN}

Joystick	Keyboard	Result
↖ ↑ ↗	U I O	
←push→	J L	movement
↙ ↓ ↘	M , .	
fire button	←	shift from Erase to Draw mode and back
	£	clear screen
	f1	cycle character color
	f2	print screen (large)
	f3	cycle background color
	f4	print screen (small)
	f5	shift graphics to Text mode and back
	f6	stop program

Table 1. Instructions for use.

Table 2. Program description of VIC Doodler.

Line #	Function
1–2	Lower top of memory for 3K expansion; skip to main routine
5–14	Get keyboard or joystick input; decode into KI
5–7	Get and decode function keys; if no input, check joystick
8	Skips to keyboard section if not using joystick
9–10	Read joystick and fire button, decode, return
11–12	Loop back to read joystick, if in use
13–14	Decode movement key
20–21	Dimension array to read joystick, initialize
25–30	Basic loop; get input, execute it
25	Gets next instruction, brings back in KI
26	Cycles next entry in array B—causes flashing
30	Decision on how to handle input—0 indicates no input, so return to 25
31–43	Handle movement instructions
31–38	Change X,Y to point to next location on screen; ensure it *is* in range 0–159, 0–175
39	Stores B(0) in old location, calculates new character CO, stores present character color there
40	Calculates address BY of new location and determines which BI bit
41	Starts setup of array B—element 1 has word with bit BI off, 2 has word with bit on
42	B(0) will hold final value—bit BI on for Draw mode, off for Erase mode
44	Cycles character color, changes in present location
45–49	Print screen section
45	Sets print flag to large and skips to it
46	Sets print flag to small
47	Realigns data direction register if using joystick
48	Prints to the screen; realigns DDR if using joystick
50–51	Cycle from Draw to Erase or vice-versa
50	Flips border color, Erase-Draw flag and true setting of present bit
51	Changes border color, indicating new mode
52	Cycles to next background color
53	Goes and clears screen (using part of initialization)
54–56	Text mode section
54	Saves border, sets border yellow and fixes DDR if using joystick

Table 2 continued.

Table 2 continued.

55	Transfers to Text mode subroutine
56	Resets screen border; continues only if last character (BO) was not f6, else falls through
57–58	Restore screen, character set, DDR, clear, quit
60–64	Text mode—get character, put on screen
60	Gets character
61	Converts to ASCII value; quits if f5 or f6
62	Converts ASCII letters (lowercase) to screen codes (uppercase)
63	Slides 19 characters in top line one left
64	Copies desired character from ROM, adds color
65–68	Initialize variables, move screen, clear
79–84	Initialize program
79	Data for reading joystick
80–83	Select joystick or keyboard; align DDR if required
84	Reads joystick array; sets up keyboard array; goes to use section at 65 to initialize screen
86–97	Screen printing section
86–92	Subroutine loop to build Y$ representing one column of dots for character, starting at I2,I1
93–94	Subroutine loop to build row of characters, print
95–97	Basic loop to build whole screen, line at a time

inch) is small. So I added the variable LS%, which causes each line of characters to be processed twice (accomplished in line 96) with LS% values of one and two.

Now, instead of line 89 building Y$, lines 90 and 91 do it. Line 90 builds the top half of a stretched character. For instance, if the top dot is on, then H(0) is 1, but instead of adding 2↑0 (or 1) to C, 3 is added (2↑0 + 2↑1). In other words, the first dot is stretched vertically over two dots.

Similarly, in the second half of the line, the created character is added twice to Y$, stretching it also horizontally. The second time the line is done, line 91 takes care of the bottom half. The result is a much larger copy, although the grain is not so fine.

This program is not only powerful, but it's easy to use. My six-year-old had no trouble enjoying the drawing part, and my eight-year-old likes the colors as well as the text at the top. They use the joystick, which is a bit faster, but I prefer the keyboard.

One of my efforts is shown in the accompanying illustration. Note that I have positioned my characters so that, on a color screen, you would see a blue bird sitting in a black tree, a yellow flower on a green stem, a red heart and various other colors strewn around. It did take a while, but I think it was worth the time.

I did not add Multicolor mode, because I like the precision I can get now, but I could easily do it. Feel free to write me about that, or any other questions you might have concerning this program. R

SERPENT OF DEATH

Venomous cobras, mummified zombies and even King Tut himself are all in this Egyptian setting that will keep you hopping from pyramid to pyramid.

by Jim Hoppe

Serpent of Death is an action arcade-style game requiring timing and skill to move "King Tut" down an ancient Egyptian pyramid while avoiding the killer cobra.

Each block on the pyramid changes color and scores one point as King Tut jumps from square to square. Stomp the killer cobra and score an extra 100 points; but watch out! If the cobra bites King Tut, he has only seconds to live.

The cobra venom is inactivated by touching the mummy who appears alongside the pyramid. If you press the fire button, the mummy will carry King Tut back to the top of the pyramid.

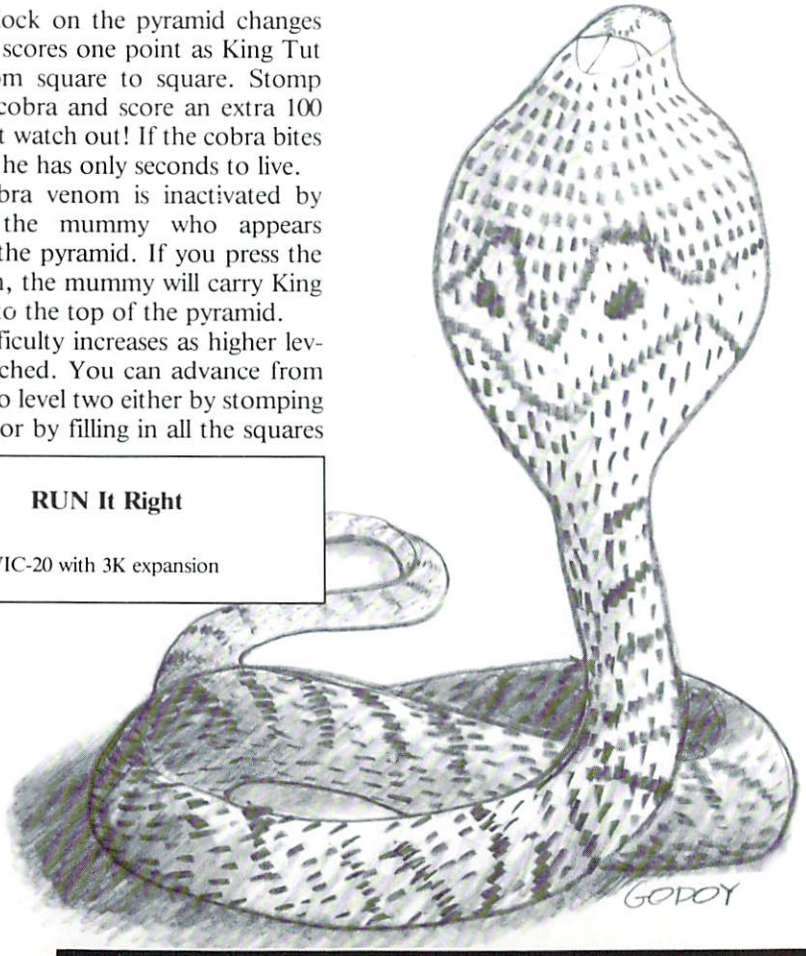
The difficulty increases as higher levels are reached. You can advance from level one to level two either by stomping the cobra or by filling in all the squares on the pyramid. Difficulty is increased at succeeding levels by requiring a greater number of cobra stomps to advance to the next level; by alternating between as many as four mummies who flash from spot to spot next to the pyramid; and by requiring all squares to be colored—as well as cobras stomped—to advance in the highest levels.

Game Design

Designing the Serpent of Death in Basic was challenging, yet fun! The initial problem in game design is coming up with a workable idea that is within the capabilities of both the machine and

RUN It Right

VIC-20 with 3K expansion



the programmer. This is critical and requires a certain amount of creativity. Many workable ideas for the beginner can be derived from already successful games.

After setting up wooden blocks on a table, I began to sketch a rough pyramid shape on paper. I broke down the basic units of the drawing in an attempt to use the VIC graphics designs already on the keyboard. It soon became evident that I'd need custom characters. Fig. 1 shows the basic building blocks of the pyramid.

Since the pyramid remains static except for color changes, I decided printing would be the easiest method for producing the design. Lines 120 through 160 accomplish this task in the program.

Using a flow chart makes life much easier in the long run. A good basic design makes it simple to modify and expand upon an idea without completely rewriting the program. My completed flow chart looked like that in Fig. 2.

Of course, a flow chart can be more detailed, but I prefer to keep mine general and fill in the details of each subroutine in Basic. If the subroutine is highly complex, a flow chart may then be required. This was the case of the jump-and-fall-to-the-square subroutine (see Fig. 3), which took me some time to perfect.

Program Details

The expression I developed for movement of characters on the screen is

```
POKE S+H+22*V, CN
POKE S+H+22*V, CC
```

where S, the starting position of Tut at the top of the pyramid, equals 7713. V = vertical position, CC = character color, H = horizontal position and CN = character Poke value. Separate variables for the cobra and King Tut make their movements independent.

The values of H and V are obtained for the joystick reading routine. For each increment of H, the horizontal coordinate increases by one (moves one space to the right). For each increment of V, the value is multiplied by 22, since moving right by 22 spaces automatically brings the character to the same horizontal position, but one row down.

The values for H and V in the cobra routine are generated randomly by lines 440–480. This makes the snake a completely independent character, jumping unpredictably all over the pyramid and thus difficult to avoid.

The background and color ahead of King Tut are set by

```
BA = PEEK (S+H+22*(V))
C1 = (PEEK (C+H+22*(V)) AND 15)
```

where C, the starting position of color at the top of the pyramid, equals 38432. S = 7713 (starting position of Tut), H = horizontal position and V = vertical position.

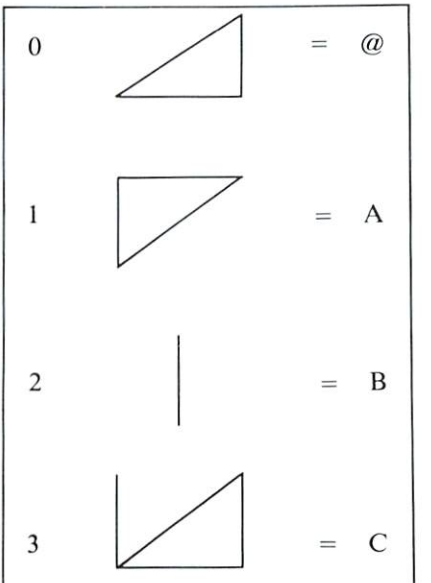


Fig. 1. The basic building blocks of the pyramid.

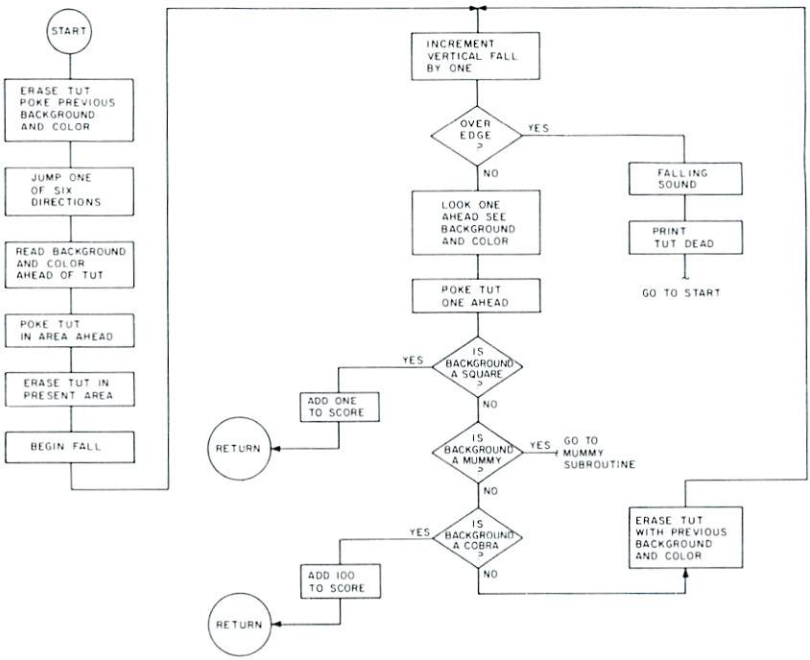


Fig. 3. Flow chart of a complex subroutine in the program.

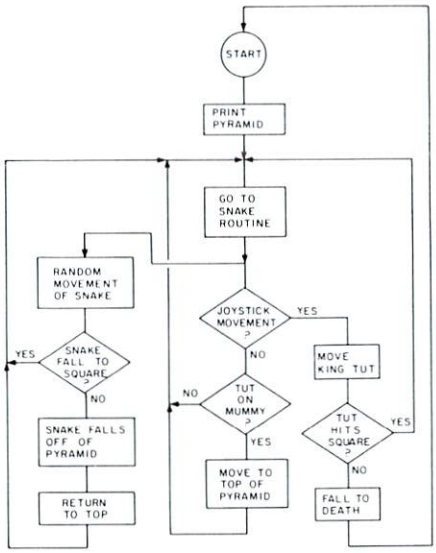


Fig. 2. Flow chart for the Serpent of Death program.

The order of events is to look at the character color and shape, memorize these values by placing them in variables, place the figure and then erase the figure with previous background and color. This sounds simple, but its proper functioning requires considerable care in design.

When Peeking color, the value must be a logical AND with 15 to filter out any values other than the basic colors 0 through 15. An originally simple concept soon becomes complex as more and more details are worked out.

As the limits of the game are reached, bugs creep in, as usual, that require program modification and change in variable values. As an example, the background color for the snake must be changed to white when a level change is made. Otherwise, the previous background color will be Poked on the top square.

I thought all the bugs were worked out of the program, only to have one of my children, a more capable games player than I, push the routines to the limit and discover a situation I had not yet provided for.

Address author correspondence to Jim C. Hoppe, S. 5309 Glenrose Road, Spokane, WA 99203.

BAJA 1000

In Baja 1000, an arcade-style game for the unexpanded VIC-20, you drive your 4×4 pickup across rough terrain and try to escape ruthless pursuers, who are in a helicopter with heat-seeking bombs.

If you can switch your exhaust from down on the ground to up in the air, you will confuse the heat-seeking capabilities of the bombs. If you can avoid the ditches and the giant cactus plants, you might make it to safety.

The space bar sends your truck into the air, which enables you to jump the ditches and the cacti. The F7 key switches your exhaust to confuse the heat-seeking bombs that are carefully dropped from the helicopter. The letter at the bottom of the screen helps you keep track of your mileage.

The course has four sections, A through D. The course listed is medium tough, and I'll teach you also how to write a more difficult one, in case this is too easy for you. (Or you may want to make it easier—for your little sister, of course.)

This program illustrates one of the most powerful features of Microsoft Basic—its ability to create graphics animation. As a beginning programmer, you soon discover that Poking your animated graphics on the screen brings the action almost to a halt.

At this point, people too often give up on Basic and turn to Forth or Assembly Code. They fail to recognize that printing strings in Basic is very close to a machine language memory move. The interpreter still has to keep track of a lot of things and do a lot of jumping around, but it's remarkably fast.

To see how fast this technique is, try the following program and refer to your manual if you don't understand how it works.

```
10 A$ = "(22 spaces) B (22 spaces)"
20 FORI = 1TO23
30 PRINT CHR$(19) MID$(A$,I,22)
40 NEXTI
50 GOTO20
```

Now add this appropriate time delay so you can see the B run across the screen:

```
35 FORT = 1TO100: NEXT
```

You might say that you could make the B go that fast by using Pokes, but remember we're not just putting up one character with this program; we're printing 22 characters, each of which could be different. We could add different colors too, and it will go just about as fast.

With Poking you need to Poke both screen and screen color memory, and if you want to shift in and out of multicolor mode, you'll see that speedwise, Poking just can't compete with printing. Try this program and then try to accomplish it with Poking.

```
10 A$ = " [color black] * [color red] *
   [color cyan] * [color purple] * [color
   green] * [color blue] * [color yellow] *
   (each color change and symbol
   should be separated by two spaces)
15 A$ = A$ + A$ + A$ + A$
20 FORI = 1TO90
25 PRINT CHR$(19) MID$(A$,I,28)
30 FORT = 1TO90:NEXT
40 NEXTI
50 GOTO20
```

RUN It Right

Unexpanded VIC-20

Address author correspondence to Bruce S. Gordon, 701 S. 11th St., Herrin, IL 62948.

This is the basic technique used to move the ground underneath the truck; a separate string moves the cacti along the outcropping of the turf. When we combine this with the mult-color graphics capabilities, you'll see that the truck wheels follow the ups and downs of the ground without a single Peek and only two Pokes.

The wheels, formed atop each piece of turf, are formed in multi-color mode in a character color the same as the screen color so that you can't see the wheel part. Then, by Poking the color memory at the location below the truck wheel-wells to a black character color, the black tires appear, and always right on the ground.

If you change POKE 646,9 in line 120 to POKE 646,8, you will see that the tires are there all the time, just invisible. Combining these two techniques allows elaborate side scrolls with reasonable speed.

The course you drive in Baja 1000 is composed of four strings—B$(1-4). These strings are scrolled with a For... Next loop beginning in line 110 and printed in line 120.

B$s are in turn composed of differing combinations of A1$, A2$ and A3$. Each B$, formed in lines 585 and 586, is made of four substrings. The substrings can be found in lines 580–583. They are composed of @, A, B, C, D and Es.

These specially created graphics characters make up the ground (and the tires). A "@" is a complete block, and an "A," an empty space. The others range in between. In these substrings, the As are the ditches, the @s will sprout cacti if there are not already cacti on the screen, and the others cause the undulations of the ground.

I encourage you to create new courses, especially after you master this one. The rules for creating the strings are simple. A1$ should start each B$. A2$ should end each B$. The first 22 characters of A1$ should be the same as the last 22 characters in A2$. I don't like the terrain too rough, so I usually just move to the next letter, and I like to avoid ditches in the first 22 characters of A1$ so the poor guy doesn't crash right away. But aside from that, you can create both easy and difficult courses. Just change those 63 character strings around and enjoy a new challenge.

The program comes in two parts and should be saved on tape, one right after the other. The first program reserves a section of memory for special characters, and then Pokes data into the memory to create the special characters. Use of special characters gives this program a high-resolution graphics appearance.

The program uses double-high characters, which are eight pixels wide and 16 pixels deep. This is the reason the data is arranged in sixteens. If you wish to vary any of the special characters—for instance, change the pickup truck into a Bronco—it should be easy to find your way. This first data statement creates the @, the second the A, the third the B and so on. The rest of the first program displays some simple directions and automatically loads the second part.

When typing in the second part of the program, you should enter line one as GOTO 510. This will prevent the program from going into double-high character mode as well as from shifting to custom characters. The program may look like the war of the alphabet, but it will let you find your errors, as the error messages come up in English instead of fragmented truck parts. When all seems to be working, then enter GOTO 500. I left out REM statements because there isn't enough memory in the unexpanded VIC, but the chart in Table 1

V— the location of the bomb
CA— the position in the cactus string
D— the control variable for the jumping truck
C— a control variable to poke out final bomb display
WC— tire color, either white or black
SN— noise voice location
SV— volume location (also auxiliary color)
D$— homes and then comes down to road level
S$— a line of solid blocks
F$— homes and then goes to the normal location of the truck
CA$— a series of cursor rights, a cactus and then more cursor rights

Table 1. Explanations of the variables used in the Baja 1000 program.

will help you.

Routines

The main game loop statements are in the first part of the program.

Lines 85–87: a subroutine that drops the bomb and checks for a hit.

Lines 200–250: the end of the game.

Lines 250–300: the Hit a Cactus routine.

Lines 300–400: the crash routine.

Lines 480–481: the data that moves the helicopter.

Lines 500 and on: just initialization.

The program is full of tricks I learned from many hours of programming bookkeeping, database and word processing programs, so it may contain some new things for game makers. It should not only be fun to play, but hopefully, it will inspire some experimentation, too. R

If any manufacturing defect becomes apparent within 30 days of purchase, the defective cassette/disk will be replaced free of charge subject to its return by the consumer by prepaid mail. Send a letter specifying the defect to:

RERUN • 80 Pine Street • Peterborough, NH 03458

Replacements will not be made if the cassette/disk has been altered, repaired, or is misused through negligence, shows signs of excessive wear or is damaged by equipment.

RERUN is simply the listing from RUN Magazine. It will not run under all system configurations. Use the Key Box accompanying each article in RUN as your guide.

The entire contents are copyrighted 1984 by CW Communications/Peterborough. Unauthorized duplication is a violation of applicable laws.

© Copyright 1984 CW Communications, Inc./Peterborough

CW COMMUNICATIONS
PETERBOROUGH
80 Pine St., Peterborough, New Hampshire 03458